Keras快速上手

基于Python的深度学习实战

谢梁 鲁颖 劳虹岚◎著

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

本书系统地讲解了深度学习的基本知识、建模过程和应用，并以深度学习在推荐系统、图像识别、自然语言处理、文字生成和时间序列中的具体应用为案例，详细介绍了从工具准备、数据获取和处理到针对问题进行建模的整个过程和实践经验，是一本非常好的深度学习入门书。

不同于许多讲解深度学习的书籍，本书以实用为导向，选择了 Keras 作为编程框架，强调简单、快速地设计模型，而不去纠缠底层代码，使得内容相当易于理解，读者可以在 CNTK、TensorFlow 和 Theano 的后台之间随意切换，非常灵活。并且本书能帮助读者从高度抽象的角度去审视业务问题，达到事半功倍的效果。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。

版权所有，侵权必究。

图书在版编目（CIP）数据

Keras 快速上手：基于 Python 的深度学习实战 / 谢梁，鲁颖，劳虹岚著.—北京：电子工业出版社，2017.8

ISBN 978-7-121-31872-6

I. ①K…II. ①谢…②鲁…③劳…III. ①软件设计 IV. ①TP311.5

中国版本图书馆 CIP 数据核字（2017）第 133701 号

策划编辑：张慧敏

责任编辑：王　静

印　　刷：三河市鑫金马印装有限公司

装　　订：三河市鑫金马印装有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱　　邮编：100036

开　　本：720×1000　1/16　　印张：16.5　字数：354 千字

版　　次：2017 年 8 月第 1 版

印　　次：2017 年 8 月第 1 次印刷

定　　价：79.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

本书咨询联系方式：（010）51260888-819　faq@phei.com.cn。

推荐语

数据挖掘与深度学习毫无疑问是大数据时代最炙手可热的研究方向。在很多前沿领域，深度学习的出现和发展正在颠覆人类对于传统计算机技术的认知。

非常有幸成为本书的首批读者，得到多位来自微软、谷歌的世界顶尖数据科学家在深度学习领域的宝贵经验分享。本书从实践角度出发，内容丰富，利用 Keras 框架讲解深度学习话题，包含了几乎全部常用的深度学习模块，并且全面、系统地介绍深度学习相关的技术，使其不再只停留于高度抽象的数学理论，具有高度的可操作性和实用性，是目前国内为数不多的中文深度学习原著之一，堪称深度学习领域的一本力作。

在此，要向深度学习领域的研究人员、算法工程师、数据爱好者强烈推荐本书，无论是初学者还是资深研究者，相信都将会从本书中获得新的收获。最后，如果有什么还需要特别强调的，那就是请深度学习这本《Keras 快速上手：基于 Python 的深度学习实战》!

亢昊辰，滨海国金所大数据中心主管

这本书自上而下地涵盖了深度学习几个最重要的方面，从软件、硬件的设置到数据的采集，从深度学习理论的介绍到实际案例的分析。整本书非常实用，讲解深入浅出，也非常高效，对于对深度学习感兴趣的读者是一本难得的好书！

周仁生，Airbnb 资深数据科学家

很久没有潜心研读一本专业书籍了。这次有机会读到这本关于深度学习的新作让我受益匪浅。深度学习近年来发展迅猛，关于这个热门课题的学习书籍和网上课程举不胜举。但是作为一个从事数据科学工作多年的统计人员，我很难找到一本深度学习的入门教程或指导书籍让我在短时间能做到理论和实践相结合。然而本书针对不同专业背景的读者，通过通俗易懂的实践和应用入手，最终把读者带到一个自己可以实战的深度学习场景。值得一提的是，对比多数关于数据科学和深度学习的书籍，这本书里的 Python 代码完整，注释详尽，而且章节之间的逻辑关系严谨。希望读者能像我一样，在有限的时间内，通过这本书能够系统掌握深度学习相关的理论和实战技术，在数据科学领域继续进阶。

刘松，Google 数据科学专家

深度学习和人工智能可谓是目前最火的话题之一，可是很多人感到入门太难。该书一改市面上很多深度学习书籍过于理论化的特点，突出实用性和可操作性，让读者能很快了解当前深度学习的成熟应用领域，并通过学习代码将解决方案移植到自己的应用环境中，是一本少有的深入浅出介绍深度学习模型及其应用的好书。该书介绍的Keras 深度学习框架提供了一个高度抽象的描述神经网络的环境，其计算后台可以在常用的 CNTK、Theano 和 TensorFlow 三个环境中自由切换，特别适合快速搭建可用于生产环境的深度学习模型。

罗勃，The University of Kansas, Associate Professor of ECS

这是一本少见的深入浅出介绍深度学习的入门书籍。该书理论和实践相结合，介绍了当前深度学习应用的几个主要框架和应用方向，实用性强，内容紧凑。基于 Keras 这个高度抽象的深度学习环境，全书强调快速构造深度学习模型和应用于实际业务，因此特别适合深度学习实践者和入门者学习，是一本必不可少的参考书。

郭彦东，微软研究院研究员

这是一部应用性很强的数据挖掘和深度学习入门书籍，内容涵盖了目前深度学习的研究应用发展最为快速的几大领域，介绍了自学架构——深度学习框架，以及解决实际问题的完整流程。作者均为在深度学习领域具有多年工作经验的数据科学家，本书详细介绍并客观评价了目前最为流行的几大前沿开源深度学习框架的实例及优缺点。本书理论体系完整，可读性强，内容言简意赅，文字深入浅出，实例极具代表性，对于诉求在较短时间内对数据挖掘和深度学习产生较为完整的理论认知并迅速投入应用实践的读者，是一本必备的教科书。

宋爽，Twitter 资深机器学习研发工程师

这是一本深度学习方面的非常实用的好书。这本书没有只拘泥于深度学习的一些理论和概念，而是通过一些例子来实现深度学习的具体应用。不论是对硬件、软件系统的搭建，还是对网络爬虫、自然语言、图像识别等重要领域的具体阐述，整本书都在详细讲述怎样将深度学习应用到各个领域。可以说，本书不仅让读者对深度学习的方法有具体了解，更重要的是在亲手教会读者利用深度学习解决很多实际的问题。

整本书的写作方式简洁明了，对问题的解释翔实而又不拖沓，可以看出是来自微软、谷歌的几位非常有知识的作者的经验之作。每一位希望学习和了解深度学习的读者，特别是希望能够将深度学习应用到具体问题的人，都可以从书中得到巨大的收获。

书中有很多实际例子和可以运行的代码，请读者一边阅读，一边尝试，相信这本书可以给读者带来事半功倍的效果。

张健，Facebook 资深数据科学家

深度学习和人工智能无疑是现在最热门的技术之一，很多人希望能掌握这方面的技能，但是担心门槛太高。这本书可谓是及时雨，给大家提供了非常好的入门学习资料，也是目前国内仅有的几本介绍 Keras 这个简单易用的深度学习框架的书。其内容不仅涵盖了当前深度学习的几个主要应用领域，而且实用性强，同时也延伸到相关的系统搭建、数据获取以及可预见的未来物联网方面的应用，非常值得一读。

陈绍林，小雨点网络贷款有限公司副总经理兼首席分析官

序一

在最近的几年里，深度学习无疑是一个发展最快的机器学习子领域。在许多机器学习竞赛中，最后胜出的系统或多或少都使用了深度学习技术。2016 年，基于深度学习、强化学习和蒙特卡洛树搜索的围棋程序 AlphaGo 甚至战胜了人类冠军。人工智能的这一胜利比预想的要早了 10 年，而其中起关键作用的就是深度学习。

深度学习已经广泛应用于我们的生活中，比如市场上可以见到的语音转写、智能音箱、语言翻译、图像识别和图像艺术化系统等，其中深度学习都是关键技术。同时，由于学术界和工业界的大量投入，深度学习的新模型和新算法层出不穷，要充分掌握深度学习的各种模型和算法并实现它们无疑是一件困难的事情。

幸运的是，基于各行各业对深度学习技术的需求，许多公司和学校开源了深度学习工具包，其中大家比较熟悉的有 CNTK、TensorFlow、Theano、Caffe、mxNet 和 Torch。这些工具包都提供了非常灵活而强大的建模能力，极大地降低了使用深度学习技术的门槛，进一步加速了深度学习技术的研究和应用。但是，这些工具包各有所长、接口不同，而且对于很多初学者这些工具包过于灵活，难以掌握。

由于这些原因，Keras 应运而生。Keras 可以被看作一个更易于使用、在更高层次上进行抽象、兼具兼容性和灵活性的深度学习框架，它的底层可以在 CNTK、TensorFlow 和 Theano 中自由切换。Keras 的出现使很多初学者可以很快地体验深度学习的一些基本技术和模型，并且将这些技术和模型应用到实际问题中。

本书也正是在这样的背景下产生的。它的目标读者正是那些刚刚进入深度学习领域、还没有太多经验的学生和工程师。本书的作者谢梁、鲁颖和劳虹岚分别在微软和谷歌这样的走在深度学习前沿的公司里做大数据和深度学习技术的研发，积累了很多把商业和工程问题转化成合适的模型并分析模型好坏以及解释模型结果的经验。在这本书里，他们把这些经验传授给大家，使更多的人能够快速掌握深度学习，并有效应用到商业和工程实践中。

这本书比较系统地讲解了深度学习的基本知识、建模过程和应用，并以深度学习在推荐系统、图像识别、自然语言处理、文字生成和时间序列的具体应用作为案例，详细介绍了从工具准备、数据获取和处理到针对问题进行建模的整个过程和实践经验，是一本非常好的深度学习入门书。

俞栋　博士
腾讯 AI Lab 副主任，杰出科学家
西雅图人工智能研究室负责人
2017 年 6 月 22 日于美国西雅图

序二

随着大数据的普及以及硬件计算能力的飞速提升，深度学习在过去的 5~6 年有了日新月异的发展。在一个又一个领域，深度学习展示了极其强大甚至连人类都难以企及的能力，这包括语音识别、机器翻译、自然语言识别、推荐系统、人脸识别、图像识别、目标检测、三维重建、情感分析、棋类运动、德州扑克、自动驾驶等。伴随着人工智能广阔的应用前景，科技巨擎诸如谷歌、微软、亚马逊、百度、腾讯、阿里巴巴等纷纷投入巨资，从而进一步推动了这个领域的进步。如今，已经很少有人还对人工智能能达到的高度有任何怀疑态度，取而代之的是对人类如何与机器共存的畅想和机器终有一天取代人类的担忧。

当然，如果我们现在就开始担心机器将毁灭人类，那么还是有一些杞人忧天。深度学习现在还只停留在感知（Perception）的阶段，即从原始数据进行简单的感觉和分析，但是远没有达到认知（Cognition）的阶段，即对事件进行逻辑推理和认识。深度学习的很多原理，还处在研究阶段。即使是各领域的专家，对于深度学习为什么如此有效，依然是一知半解。幸运的是，在解决很多实际问题时，其实并不需要我们那么深刻理解它。谢梁、鲁颖和劳红岚的这本书，就是从非常实用的角度来分享深度学习的一些基本知识，值得一读。

这本书从如何准备深度学习的环境开始，手把手地教读者如何采集数据，如何运用一些最常用，也是目前为止被认为最有效的一些深度学习算法来解决实际问题。覆盖的领域包括推荐系统、图像识别、自然语言情感分析、文字生成、时间序列、智能物联网等。不同于许多同类的书籍，这本书选择了 Keras 作为编程软件，强调简单、快速的模型设计，而不去纠缠底层代码，使得内容相当易于理解。读者可以在 CNTK、TensorFlow 和 Theano 的后台之间随意切换，非常灵活。即使你有朝一日需要用更低层的建模环境来解决更复杂的问题，相信也会保留从 Keras 中学来的高度抽象的角度审视你要解决的问题，让你事半功倍。

这一波深度学习的大潮，必将带来一个新的信息革命。每一次如此巨大的变革，都将淘汰很多效率低下的工作，并发展出新兴的职业。在一个如此激动人心的年代，愿这本书带着读者启航！

张察　博士

CNTK 主要作者之一，美国微软总部首席研究员

2017 年 6 月于美国西雅图

前言

2006 年，机器学习领域迎来了重要的转折点。加拿大多伦多大学教授、机器学习领域泰斗 Geoffrey Hinton 和他的学生 Ruslan Salakhutdinov 在《科学》上发表了一篇关于深度置信网络 (Deep Belief Networks) 的论文。从这篇论文的发表开始至今，深度学习有着迅猛的发展。2009 年，微软研究院语音识别专家俞栋和邓力博士与深度学习专家 Geoffery Hinton 合作。2010 年，美国国防部 DARPA 和斯坦福大学、纽约大学和 NEC 美国研究院合作深度学习项目。2011 年微软宣布基于深度神经网络的识别系统取得成果并推出产品，彻底改变了语音识别原有的技术框架。从 2012 到 2015 年，深度学习技术在图像识别领域取得惊人的效果，在 ImageNet 评测上将错误率从 26% 一路降到 5% 以下，几乎接近甚至超过人类的水平。这些都直接促进了一系列围绕深度学习技术的智能产品在市场上的出现，比如微软的认知服务（Cognitive Services）平台，谷歌的智能邮件应答和谷歌助手等。

在中国，我们同样欣喜地看到，基于大数据的机器学习和深度学习算法的大规模应用给互联网行业带来的巨大变革：淘宝的推荐算法、微软的小冰聊天机器人、百度的度秘、滴滴的预估时间和车费、饿了么的智能调度等都应运而生。我们有理由相信，未来的物联网、无人驾驶等也会挖掘出更多深度学习的实用场景。

深度学习对很多科技行业的从业者来说仍有一些神秘感。虽然像谷歌、微软等互联网巨头开源了诸如 TensorFlow、CNTK 等深度学习平台，大幅降低了从业者的门槛，但是如何举一反三，根据实际问题选择合适的算法和模型，并不容易。作为本书的作者，我们三位在美国谷歌、微软等顶尖互联网科技公司从事多年以机器学习和深度学习为基础的人工智能项目研发，有着丰富的实践经验，深感有必要撰写一本深入浅出的深度学习书籍，分享我们对深度学习的理解和想法，并帮助同行和感兴趣的朋友们快速上手，建立属于自己的端到端的深度学习模型，从而在大数据、深度学习的浪潮中有着更好的职业发展。我们希望本书能起到抛砖引玉的作用，使读者对深度学习产生更多的兴趣，并把深度学习作为一个必备的分析技能。

在本书中，我们选择 Keras 这个流行的深度学习建模框架来讲解深度学习话题。这主要从三方面的考虑。首先，Keras 包括了各种常用的深度学习模块，可以应用于绝大部分业务环境。其次，从原理上讲，它是高度抽象的深度学习编程环境，简单易学。Keras 底层是调用 CNTK、TensorFlow 或 Theano 执行计算的。最后，作为应用领域的从业者，我们需要关注的是如何把一个商业或者工程问题转化成合适的模型，如何准备数据和分析模型的好坏以及如何解释模型的结果。Keras 非常适合这样的场景，让使用者脱离具体的矩阵计算和求导，而将重心转移到业务逻辑上。

本书是目前国内不多的系统讲解使用 Keras 这个深度学习框架进行神经网络建模的实用书籍，非常适合数据科学家、机器学习工程师、人工智能应用工程师和工作中需要进行预测建模以及进行回归分析的从业者。本书也适合对深度学习有兴趣的不同背景的从业者、学生和老师。

本书分成 10 章，系统性地讲解深度学习基本知识、使用 Keras 建模过程和应用，并提供详细代码，使读者可以花最少的时间把核心建模知识学到手。其中第 1 章介绍搭建深度学习环境，是整本书的基础。第 2 章介绍如何用网络爬虫技术收集数据并使用 ElasticSearch 存储数据。因为在很多应用中，数据需要读者自行从网上爬取和并加以处理和存储。第 3 章介绍深度学习模型的基本概念。第 4 章介绍深度学习框架 Keras 的用法。第 5~9 章，是 5 个深度学习的经典应用。我们会依次介绍深度学习在推荐系统、图像识别、自然语言处理、文字生成和时间序列的具体应用。在介绍这些应用的过程中会穿插各种深度学习模型和代码，并和读者分享我们对于这些模型的原理和应用场景的体会。最后，我们抛砖引玉地把物联网的概念提出来。我们相信，物联网和深度学习的结合会爆发出巨大的能量和价值。

限于篇幅，我们无法涉及深度学习的方方面面，只能尽自己所能，和大家分享尽可能多的体会、经验和易于上手的代码。

在写书的过程中，我们得到了大量的帮助和指导。微软 CNTK 的作者、国际顶尖深度学习专家俞栋博士和张察博士为本书作序，并给予我们许多支持和鼓励。微软研究院的研究员郭彦东博士和高级工程师汤成对本书的部分章节提出了审阅意见。电子工业出版社的张慧敏、葛娜和王静老师，对书籍的出版和编辑付出了极大努力，才使这本书得以如期问世。在此一并感谢。

最后，我们三位作者希望本书能为中国的深度学习和人工智能的普及，为广大从业者提供有价值的实践经验和快速上手贡献我们的微薄之力。

谢梁，美国微软总部首席数据科学家
鲁颖，谷歌总部数据科学技术专家
劳虹岚，美国微软总部微软研究院研究工程师
2017 年 6 月于美国西雅图和硅谷

阅读须知

本书图片

部分图片可能需要放大观察，纸面上无法呈现应有效果。为此，相关图片均在博文视点官方网站提供下载。

读者服务

轻松注册成为博文视点社区用户（www.broadview.com.cn），扫码直达本书页面。

- **下载资源：**本书如提供示例代码及资源文件，均可在 下载资源 处下载。
- **提交勘误：**您对书中内容的修改意见可在 提交勘误 处提交，若被采纳，将获赠博文视点社区积分（在您购买电子书时，积分可用来抵扣相应金额）。
- **交流互动：**在页面下方 读者评论 处留下您的疑问或观点，与我们和其他读者一同学习交流。

页面入口：http://www.broadview.com.cn/31872



目录

1 准备深度学习的环境 1

1.1 硬件环境的搭建和配置选择 1

1.1.1 通用图形处理单元 3

1.1.2 你需要什么样的 GPU 加速卡 6

1.1.3 你的 GPU 需要多少内存 6

1.1.4 是否应该用多个 GPU 10

1.2 安装软件环境 12

1.2.1 所需软件列表 12

1.2.2 CUDA 的安装 13

1.2.3 Python 计算环境的安装 13

1.2.4 深度学习建模环境介绍 15

1.2.5 安装 CNTK 及对应的 Keras 17

1.2.6 安装 Theano 计算环境 23

1.2.7 安装 TensorFlow 计算环境 25

1.2.8 安装 cuDNN 和 CNMeM 27

2 数据收集与处理 28

2.1 网络爬虫 28

2.1.1 网络爬虫技术 29

2.1.2 构造自己的 Scrapy 爬虫 30

2.1.3 构造可接受参数的 Scrapy 爬虫 35

2.1.4 运行 Scrapy 爬虫 36

2.1.5 运行 Scrapy 爬虫的一些要点 38

2.2 大规模非结构化数据的存储和分析 40

2.2.1 ElasticSearch 介绍 42

2.2.2 ElasticSearch 应用实例 44

3 深度学习简介 57

3.1 概述 57

3.2 深度学习的统计学入门 58

3.3 一些基本概念的解释 61

3.3.1 深度学习中的函数类型 62

3.3.2 深度学习中的其他常见概念 65

3.4 梯度递减算法 67

3.5 后向传播算法 70

4 Keras 入门 72

4.1 Keras 简介 72

4.2 Keras 中的数据处理 73

4.2.1 文字预处理 74

4.2.2 序列数据预处理 82

4.2.3 图片数据输入 83

4.3 Keras 中的模型 83

4.4 Keras 中的重要对象 86

4.5 Keras 中的网络层构造 90

4.6 使用 Keras 进行奇异值矩阵分解 102

5 推荐系统 105

5.1 推荐系统简介 105

5.2 矩阵分解模型 108

5.3 深度神经网络模型 114

5.4 其他常用算法 117

5.5 评判模型指标 119

6 图像识别 121

6.1 图像识别入门 121

6.2 卷积神经网络的介绍 122

6.3 端到端的 MNIST 训练数字识别 127

6.4 利用 VGG16 网络进行字体识别 131
6.5 总结 135

7 自然语言情感分析 136
7.1 自然语言情感分析简介 136
7.2 文字情感分析建模 139
7.2.1 词嵌入技术 139
7.2.2 多层全连接神经网络训练情感分析 140
7.2.3 卷积神经网络训练情感分析 143
7.2.4 循环神经网络训练情感分析 144
7.3 总结 146

8 文字生成 147
8.1 文字生成和聊天机器人 147
8.2 基于检索的对话系统 148
8.3 基于深度学习的检索式对话系统 159
8.3.1 对话数据的构造 159
8.3.2 构造深度学习索引模型 162
8.4 基于文字生成的对话系统 166
8.5 总结 172

9 时间序列 173
9.1 时间序列简介 173
9.2 基本概念 174
9.3 时间序列模型预测准确度的衡量 178
9.4 时间序列数据示例 179
9.5 简要回顾 ARIMA 时间序列模型 181
9.6 循环神经网络与时间序列模型 186
9.7 应用案例 188
9.7.1 长江汉口月度流量时间序列模型 190
9.7.2 国际航空月度乘客数时间序列模型 203
9.8 总结 209

10 智能物联网 210

10.1 Azure 和 IoT 210

10.2 Azure IoT Hub 服务 213

10.3 使用 IoT Hub 管理设备概述 215

10.4 使用.NET 将模拟设备连接到 IoT 中心 218

10.5 机器学习应用实例 237

1 准备深度学习的环境

1.1 硬件环境的搭建和配置选择

从事机器学习，一个好的硬件环境是必不可少的。在硬件环境的选择上，并不是一定选择最贵的就会有最好的效果，很多时候可能付出了 2 倍的成本，但是性能的提升却只有 10%。深度学习的计算环境对不同部件的要求不同，因此这里先简要讨论一下硬件的合理搭配。如果您不差钱，则可以跳过本节。另外，虽然目前有一些云服务供应商提供 GPU 计算能力，并且一键部署，听起来不错，但是基于云计算的 GPU 实例受到两个限制。首先，普通的廉价 GPU 实例内存稍小，比如 AWS 的 G2 实例目前只支持单 GPU 4GB 的显存；其次，支持较大显存的实例费用比较高，性价比不高。比如 AWS 的 P2 实例使用支持每 GPU 12GB 内存的 K80 GPU，每小时费用高达 0.9 美元。但是 K80 GPU 属于 Kepler 架构，是两代前的技术。另外，在实际使用中需要开启其他服务以使用 GPU 实例，各种成本加起来每月的开支还是很可观的，很可能 6 个月的总开支够买一台配置较新 GPU 的全新电脑了。

在搭配深度学习机器而选择硬件的时候，通常要考虑以下几个因素。

(1) 预算。这个非常重要。如果预算足够，当然可以秉承最贵的就是最好的理念来选择。但是当预算有一定限制的时候，如何搭配部件来最大化性能，尽量减少瓶颈就是很重要的考量了。

(2) 空间。这里特指机箱的空间。大部分新的 GPU 都是双风扇的，因此对机箱尺寸要求很高。如果你已经有一个机箱了，那么选择合适尺寸的 GPU 就成为最优先的考虑；如果新配机箱，那么全尺寸的大机箱是最好的选择。这是因为大机箱通风好，同时

可以为以后添加多个 GPU 进行升级留有余地；另外，大机箱通常有多个 PCIe 的背板插槽可以放置多个 PCIe 设备。一般现在的 GPU 卡都会占据两个 PCIe 的插槽空间，因此背板插槽越多越好。

(3) 能耗。性能越好的 GPU 对能源的要求越高，而且很可能是整个系统里能耗最高的部件。如果已经有一台机器了，只是要添加一个 GPU 来做学习用，那么选择性能一般但是能耗低的 GPU 卡是比较明智的；如果需要高密度计算，搭配多个 GPU 并行处理，那么对电源的要求非常高，一般来说，搭配 4GPU 卡的系统至少需要 1600W 的电源。

(4) 主板。对主板的选择非常重要，因为涉及跟 GPU 的接口选择。一般来说，至少需要一块支持 PCIe 3.0 接口的主板。如果以后要升级系统到多个 GPU，那么还需要支持 8+16 芯 PCIe 电源接口的主板，这样可以连接最多 4 个 GPU 进行 SLI 并联。对于 4 个 GPU 这个限制，是因为目前最好的主板也只支持最多 40 条 PCIe 通道（16x, 8x, 8x, 8x 的配置）。多个 GPU 并行加速比并不能达到完美，毕竟还是有些额外开销的。比如系统需要决定在哪个 GPU 上进行这个数据块对应的计算任务。我们后面会提到，CNTK 计算引擎的并行加速性很好，在使用多个 GPU 时值得考虑。

(5) CPU。CPU 在深度学习计算中的作用不是非常显著的，除非使用 CPU 进行深度学习算法的计算。因此如果你已经有一台电脑的话，就不用太纠结是否要升级 CPU 了；但是如果要新搭建系统，那么在 CPU 的选择上还是有些考量的，这样可以使系统利用 GPU 的能力最大化。首先要选择一个支持 40 条 PCIe 通道的 CPU。不是所有的 CPU 都支持这么多的 PCIe 通道，比如 haswell 核心的 i5 系列 CPU 就支持最多 32 条通道。其次要选择一个高频率的 CPU。虽然系统使用 GPU 做具体的计算，但是在准备模型阶段 CPU 还是有重要作用的，因此选择使用在预算内主频高、速度快的 CPU 还是比较重要的。CPU 的核心数量不是一个很重要的指标，一般来说，一个 CPU 核心可以支持一块 GPU 卡。按照这个标准，大部分现代的 CPU 都是合格的。

(6) 内存。内存容量还是越大越好，以减少数据提取的时间，加快和 GPU 的交换。一般原则是按照 GPU 内存容量的至少两倍来配置主机内存。

(7) 存储系统。对于存储系统的能力，除要容量大以外，主要体现在计算时不停地提取数据供应 GPU 进行计算方面。如果做图像方面的深度学习，数据量通常都非常大，因此可能需要多次提取数据才能完成一轮计算，这个时候存储系统读取数据的能力就成为整个计算的瓶颈。因此，大容量的 SSD 是最好的选择。现在的 SSD 读取速度已经超过 GPU 从 PCIe 通道装载数据的速度。如果使用传统的机械硬盘，组成 RAID 5 也是一个不错的选择。如果数据量不是很大，那么这个考虑就不那么重要了。

(8) GPU。GPU 显然是最重要的选择，对整个深度学习系统的影响最大。相对于使用 CPU 进行计算，GPU 对于提高深度学习的速度是众所周知的事情，通常我们能见到

5 倍左右的加速比，而在大数据集上这个优势甚至达到了 10 倍。尽管好处明显，但是如何在控制性价比的条件下选择一个合适的 GPU 却不是一件简单的事情。因此，我们在下面的章节中将详细讨论如何选择 GPU。

1.1.1 通用图形处理单元

在介绍 GPU 加速卡的选择之前，我们先聊聊通用图形处理单元（GPGPU）。GPGPU 一般只用在图形计算上（以前这些计算是由 CPU 完成的）。从本质上讲，GPGPU 管道就是一个或者多个 GPU 和 CPU 之间的并行处理，它们对数据像图像或者其他图形格式一样进行分析处理。虽然 GPU 工作频率比 CPU 低，但是它们有更多的核心，所以可以更快地对图片和图形数据进行操作。把需要分析的数据转换成图形格式后再分析，可以有很可观的加速效果。

GPGPU 管道最初被开发用于更好地进行一般的图形处理，后来这些管道被发现更符合科学计算的需求，之后就朝着科学计算的方向开发出来了。

2001 年后，因为对可编程着色器和图形处理器的浮点运算的支持，在 GPU 上进行通用计算变得实用和流行起来。值得注意的是，涉及矩阵和/或向量（特别是二维、三维或四维向量）的问题很容易转化为 GPU 适合的计算。科学计算社区对新硬件的实验开始于矩阵乘法程序：在 GPU 上运行速度比 CPU 高的首选流行的科学问题之一是 LU 因式分解。

这些早期使用 GPU 作为通用处理器的努力需要根据图形处理器的 OpenGL 和 DirectX 两个主要 API 支持的图形元素重新构建计算问题。由于通用编程语言和 API（例如 Sh / RapidMind、Brook 和 Accelerator）的出现，这种烦琐的重构就不需要了。然后出现的是 NVIDIA 的 CUDA，不需要程序员考虑底层的适用于高性能计算的图形概念。同时，较新的独立于硬件供应商的编程架构，包括微软的 DirectCompute 和苹果 / Khronos 集团的 OpenCL 的出现使得软件可以方便地并行利用多核 CPU 和 GPU。这意味着现代 GPGPU 管道可以利用 GPU 的速度，而不需要将数据完全和显式地转换为图形形式。

DirectX 9 以前的图形卡仅支持调色或者整数颜色类型。每种可用的格式都包含一个红色、一个绿色和一个蓝色元素，再加上额外的 α 值，用于表示透明度。通用格式有：

- 每像素 8 位——有时候用调色板模式，其中每个值都是表中的索引，指向其他格式定义的实际颜色值。有时候 3 位为红色，3 位为绿色，2 位为蓝色。
- 每像素 16 位——通常这些位被分配为红色 5 位，绿色 6 位，蓝色 5 位。
- 每像素 24 位——红色、绿色和蓝色分别有 8 位。
- 每像素 32 位——红色、绿色、蓝色和 α 值都为 8 位。

浮点编码和功能在 2008 年最后修订的 IEEE 754 标准中进行了定义。David Goldberg 在他的论文 *What Every Computer Scientist Should Know About Floating-Point Arithmetic* 中对浮点和许多出现的问题进行了很好的介绍。

标准要求二进制浮点数据在 3 个字段上进行编码：1 位符号字段，后跟通过特定于每种格式的数字偏移编码指数偏移的指数位，以及编码有效数（或分数）的位，如图1.1所示。



图 1.1　二进制浮点数据编码

为了实现跨平台的一致性计算和交换浮点数据的需求，IEEE 754 标准定义了基本格式和交换格式。32 位和 64 位基本二进制浮点格式对应于 C 数据类型 float 和 double，它们的对应表示具有图1.2所显示的长度。



图 1.2　IEEE 754 标准定义的 32 位和 64 位二进制浮点格式

对于表示有限值的数值数据，符号是负号或者正号，指数字段编码基数为 2 的指数，分数字段编码有效数，而没有最高有效非零位。例如，值 -192 等于 $(-1)^1 \times 2^7 \times 1.5$，即同时表示为具有负号、指数为 7 和分数部分的编码。规定单精度浮点数指数的偏离量为 127，双精度浮点数指数的偏离量为 1023，以允许指数从负数延伸到正数。因此，上面例子中指数 7 的对应阶码表示，在单精度浮点数情况下为指数 7 加上 127，等于 134；而在双精度浮点数情况下则为 7 加上 1023，等于 1030。1.5 的整数部分，即 1，隐含在分数部分编码中，如图1.3所示。

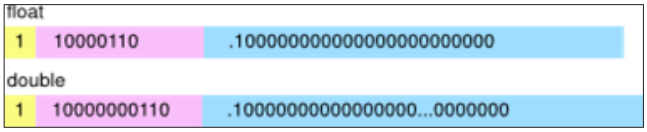


图 1.3　有限值的浮点数格式

此外，保留代表无穷大和非数字（NaN）数据的编码。IEEE754 标准全面描述了浮点编码。

假设分数字段使用有限数量的位，并不是所有的实数都能被精确地表示出来。例如，以二进制形式表示的分数 2/3 的数值为 0.10101010 ...，其在二进制点之后具有无限数量的位。值 2/3 必须首先舍入，以便以有限的精度表示为浮点数。四舍五入和舍入模式的规则在 IEEE 754 标准中有规定。最常用的是四舍五入到最近或偶数模式（缩写为

round-to-nearest）。在此模式下舍入的值 2/3 用二进制形式表示为图1.4所示的样子：符号为正号，存储的指数值表示 –1 的指数。

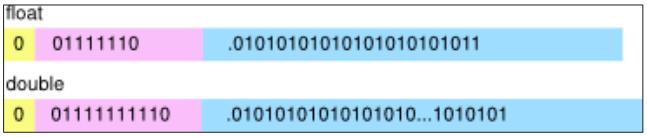


图 1.4　在 IEEE 754 标准下分数的表达

GPU 上的大多数操作都以矢量化方式运行：一次可以执行多达 4 个值。例如，如果一种颜色 <R1, G1, B1> 被另一种颜色 <R2, G2, B2> 调制，则 GPU 可以通过 <R1 * R2, G1 * G2, B1 * B2> 的向量操作由一种颜色产生另一种所需的颜色。此功能在图形中非常有用，因为几乎每种基本数据类型都是向量（二维、三维或四维）。

CPU（中央处理单元）经常被称为 PC 的大脑。但是，越来越多的 PC 正在由它的另一部分即 GPU（图形处理单元）来增强，这是其灵魂。

所有 PC 都具有将显示图像呈现给显示器的芯片，但并不是所有这些芯片都是一样的。英特尔的集成显卡控制器提供基本图形，只能显示生产力应用程序，如 Microsoft PowerPoint、低分辨率视频和基本游戏。

GPU 本身就是一个类——它远远超出了基本的图形控制器功能，是一个可编程且功能强大的计算设备。

GPU 的高级功能最初主要用于 3D 游戏渲染。但是现在，这些能力正受到更广泛的利用，比如加速金融建模、尖端科学研究和油气勘探等领域的计算工作。同时 GPU 加速计算已经成为苹果（OpenCL）和微软（使用 DirectCompute）的最新操作系统支持的主流动作。广泛接受和主流应用的原因是 GPU 计算能力强，其功能增长速度快于以 x86 为代表的传统 CPU。

在今天的 PC 中，GPU 可以承担许多多媒体任务，例如加速 Adobe Flash 视频、不同格式的视频转换、图像识别、病毒码匹配等，非常适合这类具有固有并行性的操作。因此，CPU 与 GPU 的结合可以提供最佳的系统性能、价格和功耗。从根本上讲，GPGPU 是一个软件概念，而不是硬件概念，它是一种算法，不是一个设备。然而，专门的设备设计可以进一步提高 GPGPU 管道的效率。传统上 GPGPU 管道对大量数据执行相对较少的计算，而大规模并行化、巨大的数据任务可以通过诸如机架计算（机架内部的许多类似的、高度定制的机器）的专门设置来进一步并行化，众多计算单元使用多个 CPU 来对应到更多的 GPU。比如一些比特币矿工就通过这种设置进行大量处理来挖掘比特币。

1.1.2 你需要什么样的 GPU 加速卡

现在独立的GPU加速卡从品牌来说有3种选择。首先是显卡的两个阵营，即NVIDIA和AMD；其次是Intel的Xeon Phi。

如果选择显卡的话，推荐NVIDIA。首先，使用NVIDIA的标准库可以非常容易地在CUDA上建立一个深度学习包，而AMD的OpenGL却没有那么强悍的标准库。对于非底层算法开发人员来说，现在AMD的GPU还没有好的深度学习包，只有NVIDIA有。即使将来有些OpenCL库发布了，但是从成熟度上讲，NVIDIA也还是会好很多，CUDA的GPU社区和GPGPU社区很大，而OpenGL的社区相对小。因此，在CUDA社区，已经有好的开源方案和指导意见供大家使用了。其次，现在NVIDIA在深度学习领域发展得非常好。早在2010年，NVIDIA就预言深度学习在未来10年内会越来越流行，因此投入大量资源进行这方面的研究和开发。AMD在这方面的投入相对于NVIDIA稍微落后了一点。

如果选择使用Xeon Phi，则只能用标准C语言，然后把C语言代码转化成加速的Xeon Phi代码。这个功能看起来挺有意思的，因为可以使用现在普遍存在的C语言源代码。然而，理想很丰满，现实很骨感。真正能被支持的C代码只有很少一部分，所以这个功能没有什么大用，而且大部分C代码运行很慢。Xeon Phi在社区支持方面也不是很好，比如Xeon Phi的MKL数值库和Numpy不兼容；在功能方面也不尽完善，比如Intel Xeon Phi编译器不能优化模板，也不支持GCC的向量化功能，Intel的ICC编译器并不支持全部的C++ 11的功能。另外，写完代码还没法进行单元测试。这些问题说明Xeon Phi还不是一个成熟、可靠的工具，不适合用来帮助一般程序员或者数据科学家进行深度学习的工作。

因此，当前最适合的选择是NVIDIA旗下的各种GPU加速卡。

1.1.3 你的 GPU 需要多少内存

锁定品牌以后，还需要选择正确的GPU型号，这时候你需要了解用多少内存跑深度学习。接下来我们讨论卷积网络的内存需求，这是因为卷积神经网络在计算的时候对内存的需求非常大，一般一块能满足训练卷积神经网络任务的GPU加速卡也能满足大部分其他计算任务。这样可以确保买到容量合适的GPU卡，而不会花冤枉钱去买高端的加速卡，但是又发挥不出全部效能。

卷积神经网络对内存的要求和简单神经网络非常不一样。如果只是存储卷积神经网络，那么所需的内存会稍小一些，因为参数少；但是如果要训练一个卷积神经网络，

那么所需的内存则非常大。这是因为每个卷积层的激活函数量和误差量相比于简单神经网络非常大，这些是内存消耗的主要来源。把激活函数量和误差量加起来可以决定大致的内存需求。然而，在这个网络里通过某一种状态确定激活函数和误差的数量是很难的。一般来说，最开始的几层会吃掉很多内存，因此主要内存需求取决于输入数据的大小。所以首先应考虑输入数据大小。

举例说明。假如需要训练一个图像识别模型，在数据中每个图片宽度为 512 像素，高度也为 512 像素，每个像素 3 个颜色通道的图像，即每个图片可以存为一个 512×512×3 的多维矩阵。再假如设定一个 3×3 的滤子，表 1.1 给出了一个 7 层的卷积神经网络所需的神经元个数，以及每个神经元对应的权重数。

表 1.1　卷积神经网络参数计算表

层数	图像高度	图像宽度	深度	滤子高度	滤子宽度	神经元数	单神经元参数
1	512	512	3	3	3	786432	108
2	256	256	6	3	3	393216	216
3	128	128	12	3	3	196608	432
4	64	64	24	3	3	98304	864
5	32	32	48	3	3	49152	1728
6	16	16	96	3	3	24576	3456
7	8	8	192	3	3	12288	6912
						1560576	13716

表 1.1 显示卷积神经网络对内存的需求是巨大的。对于一个 115 × 115 × 3 的图像，如果所有参数都以 8 位的浮点数来存储的话，那么最终所需的内存则高达 1GB。

当然，在实际应用中，可以根据硬件的 GPU 内存限制来缩小计算的规模。比如在常见的 ImageNet 数据集中，通常使用 224×224×3 的多维矩阵存储图像。假如训练这样的一个数据集可能需要 12GB 的内存，如果将图像缩小为 115 × 115 × 3 的维度，那么只需要四分之一的内存容量就够训练模型了。

此外，GPU 的内存需求也取决于数据集的样本量。比如只用了 ImageNet 数据集的 10%，那么一个非常深的模型就会很快地过度拟合，因为没有足够的样本泛化。但是较小的神经网络通常会表现比较好，同时所需的内存也较少，这样 4GB 或者更少的内存就足够用了。这表明用较少的图片来训练模型时，因为模型复杂度降低了，所以对内存的需求就会少些。

第三个决定内存需求的因素是分类类别的数目。相比训练一个 1000 种类别图像的模型，如果只是训练一个二分类图像的问题，那么所需的内存会小很多。这是因为如果

有更少的类需要互相区别的话，则过度拟合发生更快，或者换句话说，比起区分 1000 个类，只需要更少的参数来区分两类。

对于卷积神经网络而言，可以使用两种方法来降低 GPU 的内存需求。第一种是采用较大的递进步数作为卷积内核，这时不是为每一个像素，而是为两个或四个像素（2 或 4 步幅）来应用卷积内核，这样将产生更少的输出数据。这个技巧通常用于输入层中，因为这一层最消耗内存。第二种是引入一个 1×1 的卷积内核层，从而降低所需的深度。例如，64×64×256 的输入可以通过 96 个 1×1 颗粒减少到 64×64×96 的输入。

最后，总是可以降低训练批量的大小。训练批量的大小对内存需求的影响非常显著。比如对于同一个模型，使用 64 个而不是 128 个训练批量，可以降低一半的内存消耗。但是，模型训练也可能需要更长的时间，特别是在训练的最后阶段。最常见的卷积操作一般是对 64 个或更大的训练批量大小进行优化，这就使得从 32 个训练批量这个大小开始，模型训练速度会大大降低。所以收缩训练批量的大小，甚至减少到 32 个以下只应作为最后手段的一个选项。

另一个经常被忽视的选择是改变卷积网络所采用的数据类型。通过从 32 位切换到 16 位，可以很容易地将内存消耗降低 50%，同时又不会显著降低模型的性能。这种方法在 GPU 上通常会得到非常明显的加速比。

所以，这些内存减少技术在面对真实数据时会是什么样的情况呢？

如果把 128 个批量大小、250 像素 × 250 像素 3 种颜色（250×250×3）图像作为输入，使用 3×3 内核，按照 32, 64, 96 步骤递增的话，那么只是为了进行误差计算和激活函数所需的内存大小为：92MB→1906MB→3720MB→5444MB。

在一块普通的 GPU 卡上，内存很快就会出现不足。如果用 16 位代替 32 位，这个数字会减半；对于训练批量个数为 64 的同理。如果同时使用 64 个训练批量和 16 位精度，内存消耗就会降为原来的四分之一。不过，如果要用多层训练深度网络，内存仍然会非常吃紧。如果在第一层添加步长 2，接着使用 2×2 的最大池化技术，那么内存消耗的变动则为：92MB (input)→952MB (conv)→238MB (pool)→240MB (conv)→340MB (conv)。

这就大幅降低了内存消耗。但是如果数据够大、模型够复杂，还是会有内存问题的。如果用了 2~3 层，那么可以用另外一层做最大池化或者应用其他技术。比如 32 个 1×1 的内核会把最后一层的内存消耗从 340MB 降低到 113MB，这样我们就可以简单地把神经网络扩展到很多层，而且不用担心性能问题。

如果使用了最大池化、跨越式和 1×1 的内核等技术，虽然这些技术可以很有效地降低内存消耗，但是也相应地扔掉了这些层的很多信息，这会对模型的预测性能不利。其实训练卷积网络的本质就是把混合的不同技术用到一个网络上，用尽量低的内存消耗得到尽量好的结果。

事实证明，最重要的实际 GPU 性能指标应该是内存带宽，即以 GB/s 衡量内存每秒读取和写入的吞吐量。内存带宽非常重要，因为目前几乎所有的数学运算如矩阵乘法、点积、求和等都受到带宽的约束，瓶颈在于多少数据可以从内存中提取出来以供运算，而不是有多少运算力可用。笔者在实际应用中使用 GTX 1060 显卡做 GPU，通常只能达到 40% 的饱和计算能力，这就是由于内存带宽的限制造成的。表1.2列出了在 Kepler、Maxwell 和 Pascal 架构下不同 GPU 的内存带宽。

表 1.2　不同 GPU 的带宽比较（GB/s）

架构	Tesla P100 (16GB)	Tesla P100 (12GB)	GTX 1080Ti	GTX 1080	GTX 1070	GTX 1060
Pascal	720	540	484	320	256	192
架构	Tesla M60	Tesla M40	GTX 980Ti	GTX 980	GTX 970	GTX 960
Maxwell	2x160	288	330	224	196	120
架构	Tesla K80	Tesla K40	GTX 690	GTX 680	GTX 660Ti	GTX 660
Kepler	2x240	288	2x256	256	192	192

在同一个架构下，带宽可以直接比较。例如同为 Pascal 架构的 GTX 1080 和 GTX 1070 高性能显卡，可以直接通过查看内存带宽相比较。然而，在不同的体系结构中，由于不同的架构如何利用给定的内存带宽不同，就不能直接通过比较内存带宽来比较性能了。例如 Pascal 架构的 GTX 1080 和 Maxwell 的 GTX Titan X 就不能直接通过比较带宽来考察其性能的差异。这使得一切都有点棘手，因为总体带宽仅会提供一个 GPU 速度的大致概念。一般在预算确定的情况下，可以选择能够买到的架构最新、内存带宽最大的显卡。显卡价格下降得非常快，因此买一块二手显卡应该是预算紧张时的最佳选择。现在 GTX 10 系列新一代显卡大行其道，因此 9 系列，特别是 960 系列显卡的价格下降较快，不失为一个入门选择。如果添加一点预算，则可以选择 1060 系列的入门显卡，价格和 980 系列差不多，但是架构更新、能耗更低、内存更大，只是带宽稍微小了点。

另一个需要考虑的重要因素是，并非所有的架构都与 cuDNN 兼容。比如 Kepler 架构的 GPU 就不行，不过它已经非常老了，估计市面上不多见了。GTX 9 系列或者 10 系列都能很好地利用 cuDNN 的计算能力，所以现在在选购时基本没有什么问题。

基于上面的带宽论述，表1.3粗略地比较了不同 GPU 执行深度学习任务时的相对性能。注意，这只是大概的比较，实际的速度对比会有一些不同。在表1.3中，Pascal 架构的 Titan X 作为目前最强的主流 GPU 之一，被设定为比较对象，标为其他主流的几种 GPU 的一个倍率，从而让读者对不同 GPU 的选择有一个比较直观的概念。例如 Pascal 架构的 Titan X = 1，而 GTX 1080 = 1.43，就表示 Pascal 架构的 Titan X 在执行深度学习任务时其速度是 GTX 1080 的 1.43 倍左右，换句话说，就是 Pascal 架构的 Titan X 比 GTX 1080 快 43%。

表 1.3　主流 GPU 的相对计算能力

Pascal Titan X	GTX 1080	GTX 1070	GTX TitanX	GTX 980Ti	GTX 1060	GTX 980
1	1.43	1.82	2.00	2.00	2.50	2.86
GTX 1080	GTX 970	GTX Titan	AWS GPU Instance	GTX 960		
1	3.33	4.0	5.72	5.72		

一般来说，如果预算比较充裕，GTX 1080 或 GTX 1070 是不错的选择。如果预算不是问题，显然 GTX 1080Ti 12GB 版本是非常好的选择；如果预算稍微紧张些，那么 GTX 1070 8GB 版本应该是性价比最高的 GPU。普通的 GTX Titan X 卡因为使用 8GB 内存，比只有 6GB 内存的 GTX 980Ti 更适合深度学习。

GTX 1060 是最好的入门 GPU。价格便宜，内存也高达 6GB，对于大多数的学习项目够用了。其主要问题是带宽接口较小，只有 192 bit，整个内存的吞吐量相对于 GTX 1080 和 1070 低了很多。不过，虽然 GTX 1060 与 1080、1070 和 Titan X 这类相对高端的 GPU 相比性能还是有所欠缺的，但是跟 GTX 980 的性能几乎持平，内存还大 2GB，因此性价比非常之高，是个人练习深度学习非常好的选择。6GB 和 8GB 内存对于大多数中等规模的深度学习任务是够用的，但是遇到 ImageNet 这种规模的数据或者视频数据就明显不够用了。这时候 12GB 内存的 GTX 1080Ti 就派上用场了。

1.1.4　是否应该用多个 GPU

通过 SLI，GPU 的互联速度达到了 40Gb/s，并且在游戏里搭建 SLI 并联显卡能显著提升性能。那么能不能通过搭建多个 GPU 来提升工作站的深度学习计算能力呢？如果能，是不是越多越好呢？图1.5展示了这样一类配置。



图 1.5　三块 GXT Titan 卡和一块 InfiniBand 卡（这是为深度学习配置的）

跟游戏里的表现相反，在多个并联的 GPU 上并行计算神经网络是很难的，而且在高密度的神经网络上获得的提升优势非常有限。对于小的神经网络，数据并行更有效；而对于大的神经网络，由于数据传输上的瓶颈，简单并联多个 GPU 有时并不能获得理想的速度提升比。

另外，在某些特定问题上 GPU 的并行效果不错。例如，卷积层可以很容易并行化，而且可以很好地进行伸缩。许多现成的框架，比如 TensorFlow、Caffe、Theano 和 Torch，都支持这种并行性，如果使用 4 个 GPU，则会看到 2.5~3 倍的速度提升；而微软的 CNTK 提供了最佳的并行性能，在很多情况下加速比能提高 3.6~3.8 倍。CNTK 使用自己专门的 BrainScript 来描述神经网络，对于初学者而言，学习曲线稍陡，不过现在也提供了基于 Python 以及其他语言的 API，方便了使用者。TensorFlow 和 Theano 都使用函数形式的 API 来描述神经网络。

目前 CNTK 和 TensorFlow、Theano 一样，也纳入了 Keras 的后端平台，用户只需了解 Keras 的高度抽象的构造方法即可，而不需要考虑很多 CNTK 与其他软件包不同的地方，可以大大提高生产效率。这也是我们写作本书的目的。

我们相信随着软件和硬件性能的提高，使用多个 GPU 并行计算神经网络来大幅提高计算性能的情况会越来越普遍，最终会让普通用户实现快速训练许多不同深度学习模型的能力。

当然，从另一个角度而言，使用多个 GPU 还有一个优点，就是可以在每个 GPU 上各自运行自己的算法或实验。虽然性能没有提升，但是同时调试不同的算法或参数，可以尽快地获得所需的理想模型。无论是对于研究人员还是数据科学家，这都是非常重要的，因为在实际工作中大部分时间其实都投入到调参中了。

另外，从心理上有助于读者保持学习的兴趣。一般来说，执行任务的时间和接收反馈的时间间隔越短，人们越能更好地将相关的印象和学习经验集成总结到一起，也不会因为长时间反复等待所需的计算结果而灰心丧气。如果读者用两个 GPU 训练两个卷积网络的小数据集，则能更快地了解什么参数对模型的效果影响大，也会更容易检测交叉验证错误的规律。这些经验的有效总结可以帮助分析师正确地理解到底需要调整哪些参数或删减哪些层来提高模型的效能。

所以，总的来说，一个 GPU 应该满足几乎所有任务，但使用多个 GPU 正变得越来越重要，有助于加快深度学习模型的建模过程。所以，如果读者想快速地训练深度学习模型，使用多个廉价的 GPU 会表现不错。

1.2 安装软件环境

下面我们准备安装深度学习的软件环境。这是本书应用案例的运行环境，对于大部分中小型的深度学习项目也是够用的。考虑到读者的背景广泛，我们选择在 Windows 系统环境下配置深度学习的软件环境。Windows 系统的版本号是 10.0.14393。

1.2.1 所需软件列表

下面是所需的软件工具和计算库列表。

(1) Visual Studio 2013 Community Edition，版本号 12.0.31101.00 Update 4。我们需要 VS 的 C/C++ 编译器和 SDK，其自带的.NET 环境为 4.6.01586。

(2) Anaconda 计算环境。对于基于 Python 的科学计算环境，现在通常的做法是安装预先打包好的 Python 科学计算环境，目前常见的有 WinPython、Anaconda Python 等。在本书中，我们选择 Anaconda 环境。Anaconda 是一个非常流行的基于 Python 的科学计算环境，预先整合了 150 多个数据科学计算库，使用非常方便，并且 Anaconda 的服务器支持超过 700 个常用的软件包额外供分析师和数据科学家下载安装和使用。虽然目前最新版本是支持 Python 3.6 的 Anaconda 3-4.4.0，但是本书使用的是基于 Python 3.5 的 Anaconda 3-4.2.0 版本，因此请读者下载 Python 3.5 对应的版本，以便顺利安装所需的计算环境。

(3) CUDA 8.0.44 (64-bit)。这是 NVIDIA 的 GPU 计算数学库和编译器。

(4) MinGW-w64 (5.4.0)。我们需要 MinGW 的编译器和工具，比如 g/g++、make 等。

(5) CNTK 2.0。CNTK 是微软开发的一个深度学习计算环境，具有速度快、GPU 并行扩展能力强等优点，也是目前在循环神经网络这类模型中计算速度最快的深度学习环境。

(6) Theano 0.9.0。Theano 是蒙特利尔大学开发并开源的一个深度学习环境，提供了对神经网络模型数学公式和多维矩阵进行代数运算的环境。

(7) TensorFlow。TensorFlow 是 Google 开发并开源的一个深度学习环境，在国内名气较大。

(8) Keras 1.1.0 或者微软自己做的服务端仓库克隆（fork）的 Keras（基于 Keras 2.0 界面）。Keras 是以 CNTK、Theano 或者 TensorFlow 为计算后台的深度学习建模环境。这个库将繁杂的数学运算抽象出来，让用户能集中精力构造自己的神经网络模型和进行建模，非常高效、方便，是本书案例的主要工具和讲解的对象。

(9) OpenBLAS 0.2.14。这个库提供了针对不同 CPU 架构优化后的线性代数数值计算库。

(10) cuDNN v5.1 (August 10, 2016) for CUDA 8.0（可选项）。这是针对卷积神经网络模型优化的数值计算库。使用该计算库，卷积神经网络的计算速度能提高 2~3 倍，非常可观。

1.2.2 CUDA 的安装

首先，我们需要安装 CUDA。可以通过如下步骤进行安装。

(1) 下载并安装 Visual Studio Community 2013。需要登录 Visual Studio 网站，如果你还没有注册，请注册一个账户登录。不要下载最新的 Visual Studio 2015，这个版本不支持 CUDA 8.0。如果你使用的是 NVIDIA 10 系列显卡，则需要使用 CUDA 8.0 驱动。

(2) 把 C:\Program Files (x86)\Microsoft Visual Studio 12.0\VC\bin 目录添加到当前可搜索路径 PATH 里面。

(3) 下载并安装 CUDA 8.0。需要登录 CUDA 网站，如果你还没有注册，则需要注册一个账户登录。安装完毕以后，可以通过运行一个样本程序来检验。选择这个样本文件，单击鼠标右键，选择 Debug→Start New Instance 开始运行这个 VS 项目，你将在屏幕上看到一个模拟的海洋洋面。

1.2.3 Python 计算环境的安装

下面我们继续安装 Python 计算环境。

(1) 下载并安装 Anaconda 4.2.0（https://repo.continuum.io/archive/Anaconda3-4.2.0-Windows-x86_64.exe，MD5=0ca5ef4dcfe84376aad073bbb3f8db00），该版本支持 Python 3.5。这个过程会持续几分钟到二十几分钟。通常安装目录选择 C:\Anaconda3\就行。安装完成以后，单击“开始”按钮可以看到几个相关的项目出现在“最近添加”项目里，如图 1.6 所示。

单击“Anaconda Prompt”，会跳出 Anaconda Console，输入 python 就能进入 Python 环境，出现如图1.7所示的提示。

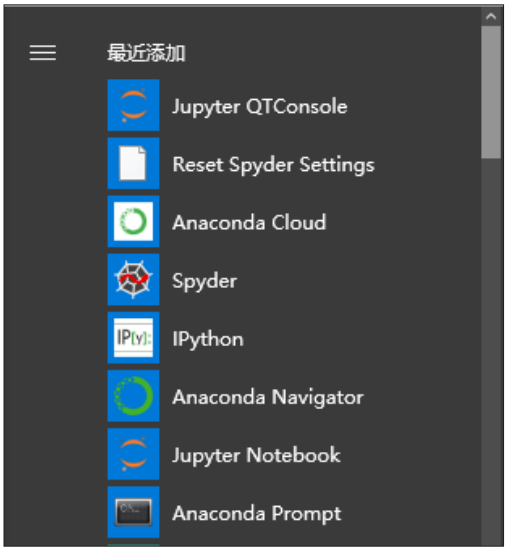


图 1.6　Anaconda 安装完成后新添加项目

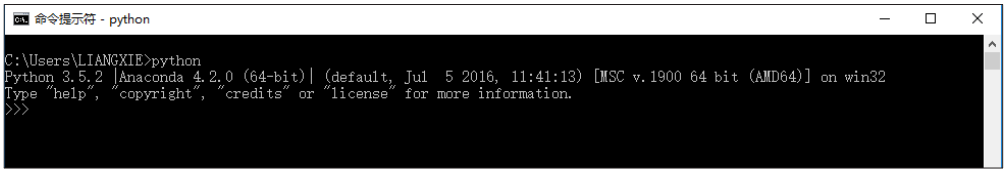


图 1.7　Python Console 显示环境

(2) 安装 MinGW 和 LibPython 包。在 4.2 版本以前的 Anaconda 环境中是无法通过 conda 安装 LibPython 包的，会提示冲突，但是在 4.2 版本中已经修正了这个问题，可以顺利安装。只需在 Anaconda Console 中输入如下命令：

```
conda install -c anaconda mingw libpython
```

即可顺利安装 MinGW 和 LibPython 这两个软件包。这是接下来安装 GPU 计算环境的前提。在安装过程中你会看到如图1.8所示的信息。

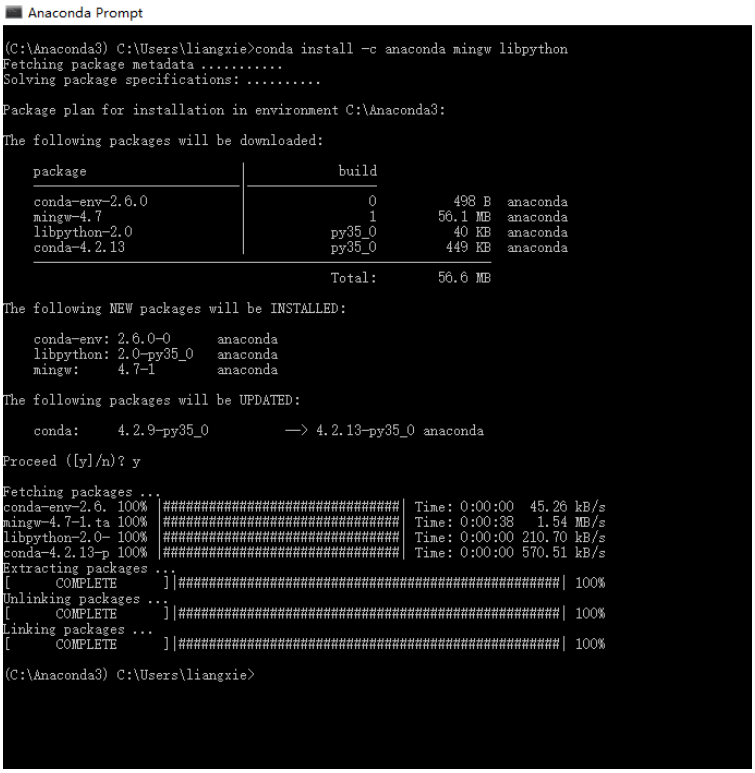


图 1.8　MinGW 和 LibPython 包安装过程

1.2.4　深度学习建模环境介绍

接下来还需要考虑 GPU 建模环境。我们选择以 Keras 为基础的 GPU 建模环境，而 Keras 是以 CNTK、Theano 或者 TensorFlow 之一作为实际后台的建模环境。Keras 相对于标准的 GPU 建模环境，比如 CNTK、Theano、TensorFlow、Caffe 等，有一个非常巨大的优势就是使用简单，将用户从繁杂的数学公式命令中解放出来，直接考虑具体的深度学习神经网络的架构。当然，缺点就是只能使用目前已有的结构来搭建自己的神经网络，并且它只提供一个建模环境，实际计算还需要依赖 CNTK、Theano 和 TensorFlow 三个后台计算环境之一来进行。不过，对于实际应用来说这已经足够了，新的未经过验证的网络结构反而在实践中应该避免应用。

下面分别介绍 CNTK、Theano 和 TensorFlow 后台的安装。

虽然目前在国内 TensorFlow 很流行，但是在本书中实际例子的计算使用的后台都是 CNTK，这是因为作为后台，CNTK 相对于 Theano 和 TensorFlow 两个后台以及其他深度学习环境具有如下优势。

(1) CNTK 在速度上有比较明显的优势。根据香港浸会大学（HKBU）的 Shaohuai Shi 等人 2017 年发表的研究结果（http://dlbench.comp.hkbu.edu.hk/），运行 GPU 时，CNTK 在大多数模型中的速度表现都优于其他计算环境，而在循环神经网络模型中更是大幅度领先于其他计算环境，这在语音识别、自然语言理解、翻译以及时间序列预测类的任务中优势很明显。在我们的试验中，CNTK 作为 Keras 的后台，在循环神经网络模型中，比 TensorFlow 平均快 30%~40%，只在简单的全链接网络模型中稍慢于 TensorFlow。测试平台是 Intel Xeon CPU 5E-2620 V2 @2.1GHz + 32GB 内存 + Windows 10 企业版。显卡选择的是 NVIDIA Titan Xp。我们选择了 8 个例子来比较 TensorFlow、CNTK 和 Theano 的性能，结果如表1.4所示。

表 1.4　三种 Keras 后台的速度比较

示例程序	迭代次数	TensorFlow	CNTK	Theano
mnist_cnn.py	12	116	94	128
mnist_mlp.py	100	485	530	569
imdb_lstm.py	15	3645	2505	3456
imdb_cnn.py	2	28	27	29
mnist_hierachical_rnn.py	5	1757	1163	1834
addition_rnn_lstm.py	200	3486	3349	3479
imdb_cnn_lstm.py	2	210	155	246
lstm_text_generation.py	10	1920	1190	2189

(2) CNTK 的预测精度很好。CNTK 提供了很多先进的算法实现来帮助提高预测精度。比如 CNTK 的 Automatic Batching Algorithm 允许分析师把不同长度的序列合并在一起，在提高运行效率的同时，实现了更优化的随机性，通常能提高 1~2 个百分点的预测精度。微软研究院通过这项技术实现了和人一样的实时对话语音识别能力（Human Parity）。

(3) CNTK 的产品质量更好。很多深度学习计算环境虽然声称能够重复论文里的样本模型结果，但是在实际运算中会遇到各种各样的问题，要么最后的结果达不到论文里标识的精度，要么就是 bug 不断，无法运行样本代码。CNTK 的质量保证对于任意模型，根据数据和模型的描述从头建模，都能够重复论文里的结果，这对于生产系统来说非常重要。比如 Inception V3 网络模型（https://arxiv.org/abs/1512.00567），这个模型如果使用别的深度学习框架的样本代码自己用数据从头训练模型的话，要么会遇到 bug，要么很难达到论文里展示的精度，因为有很多数据的预处理等小技巧没有在样本代码里提供。如果使用 CNTK 的样本代码就能顺利地实现模型的训练并达到优于论文里的预测精度。感兴趣的读者可以去 https://github.com/Microsoft/CNTK/tree/master/Examples/ 下载不同的样本代码来学习。

(4) CNTK 在 GPU 上的扩展性很好。真正的生产用深度学习模型通常需要极大量的数据，因此在实际训练时需要尽可能多的 GPU。CNTK 可以很轻松地并行到数百个 GPU 上，特别是 CNTK 独家的 1-bit SGD 和 Block-Momentum SGD 算法，可以实现高度的并行计算。2014 年微软发表于 *INTERSPEECH* 上的论文显示，使用 1-bit SGD 算法，针对不同规模的数据和迷你批量数，使用 8 个 K20X GPU 能在常见的 Switchboard DNN 上实现 3.6~6.3 倍的加速能力，其中较大的迷你批量数能实现较高的倍增率。而根据 2016 年微软发表于 *ICASSP* 上的论文，CNTK 的 Block-Momentum SGD 算法在 LSTM 和 DNN 类型的模型中能实现近乎线性加速能力。

(5) CNTK 的 API 设计基于 C++，在速度和可用性上很好。因为 CNTK 的所有核心功能都是基于 C++ 的，因此速度自然有保证，同时在其他语言中写接口也非常自然、方便。虽然本书是教读者使用 Keras 来建模，但是如果后台使用 CNTK 的话，则可以保证分析环境和生成环境的结果相同。同时 CNTK 的 Python API 有高抽象版本和低抽象版本，其中高抽象版本基于函数式编程（Functional Programming）的理念，使用起来非常方便；而低抽象版本则适合应用于生产系统中。

1.2.5 安装 CNTK 及对应的 Keras

下面介绍 CNTK 2.0 版本的安装。

首先打开 Anaconda Console，建立一个新的 Anaconda 虚拟环境，这样就不会和已经安装好的 Python 包冲突了。输入以下命令：

```
(d:\Program Files\Anaconda3) E:\code> conda create --name cntkKeraspy35
Python=3.5 numpy scipy h5py jupyter
```

如果读者有其他常用的库，比如 pandas、matplotlib 等，则可以在后面添加上。然后就会看到下面的屏幕输出，表明 Anaconda 在新创建一个虚拟环境，并安装指定的软件包。

```
1  Fetching package metadata ...........
2
3  Solving package specifications: .
4
5  Package plan for installation in environment d:\Program Files\Anaconda3\envs
   \cntkKeraspy35:
6
7  The following NEW packages will be INSTALLED:
```

```
8
9       bleach:               1.5.0-py35_0
10      colorama:             0.3.9-py35_0
11      decorator:            4.0.11-py35_0
12      entrypoints:          0.2.2-py35_1
13      h5py:                 2.7.0-np112py35_0
14      hdf5:                 1.8.15.1-vc14_4     [vc14]
15      html5lib:             0.999-py35_0
16      icu:                  57.1-vc14_0         [vc14]
17      ipykernel:            4.6.1-py35_0
18      iPython:              6.0.0-py35_1
19      iPython_genutils:     0.2.0-py35_0
20      ipywidgets:           6.0.0-py35_0
21      jedi:                 0.10.2-py35_2
22      jinja2:               2.9.6-py35_0
23      jpeg:                 9b-vc14_0           [vc14]
24      jsonschema:           2.6.0-py35_0
25      jupyter:              1.0.0-py35_3
26      jupyter_client:       5.0.1-py35_0
27      jupyter_console:      5.1.0-py35_0
28      jupyter_core:         4.3.0-py35_0
29      libpng:               1.6.27-vc14_0       [vc14]
30      markupsafe:           0.23-py35_2
31      mistune:              0.7.4-py35_0
32      mkl:                  2017.0.1-0
33      nbconvert:            5.1.1-py35_0
34      nbformat:             4.3.0-py35_0
35      notebook:             5.0.0-py35_0
36      numpy:                1.12.1-py35_0
37      openssl:              1.0.2k-vc14_0       [vc14]
38      pandocfilters:        1.4.1-py35_0
39      path.py:              10.3.1-py35_0
40      pickleshare:          0.7.4-py35_0
41      pip:                  9.0.1-py35_1
42      prompt_toolkit:       1.0.14-py35_0
43      pygments:             2.2.0-py35_0
44      pyqt:                 5.6.0-py35_2
```

```
    Python:               3.5.3-3
    Python-dateutil:      2.6.0-py35_0
    pyzmq:                16.0.2-py35_0
    qt:                   5.6.2-vc14_4        [vc14]
    qtconsole:            4.3.0-py35_0
    scipy:                0.19.0-np112py35_0
    setuptools:           27.2.0-py35_1
    simplegeneric:        0.8.1-py35_1
    sip:                  4.18-py35_0
    six:                  1.10.0-py35_0
    testpath:             0.3-py35_0
    tornado:              4.5.1-py35_0
    traitlets:            4.3.2-py35_0
    vs2015_runtime:       14.0.25123-0
    wcwidth:              0.1.7-py35_0
    wheel:                0.29.0-py35_0
    widgetsnbextension:   2.0.0-py35_0
    win_unicode_console: 0.5-py35_0
    zlib:                 1.2.8-vc14_3        [vc14]

Proceed ([y]/n)? y

mkl-2017.0.1-0 100% |################################| Time: 0:00:02  56.40
MB/s
jpeg-9b-vc14_0 100% |################################| Time: 0:00:00  15.43
MB/s
openssl-1.0.2k 100% |################################| Time: 0:00:00  30.41
MB/s
Python-3.5.3-3 100% |################################| Time: 0:00:01  30.70
MB/s
colorama-0.3.9 100% |################################| Time: 0:00:00   0.00
B/s
decorator-4.0. 100% |################################| Time: 0:00:00   2.07
MB/s
entrypoints-0. 100% |################################| Time: 0:00:00 599.98
kB/s
iPython_genuti 100% |################################| Time: 0:00:00   2.51
```

```
MB/s
75 jedi-0.10.2-py 100% |################################| Time: 0:00:00  16.38
MB/s
76 jsonschema-2.6 100% |################################| Time: 0:00:00   0.00
B/s
77 libpng-1.6.27- 100% |################################| Time: 0:00:00  32.88
MB/s
78 mistune-0.7.4- 100% |################################| Time: 0:00:00   9.32
MB/s
79 numpy-1.12.1-p 100% |################################| Time: 0:00:00  32.43
MB/s
80 pandocfilters- 100% |################################| Time: 0:00:00   0.00
B/s
81 path.py-10.3.1 100% |################################| Time: 0:00:00   0.00
B/s
82 pygments-2.2.0 100% |################################| Time: 0:00:00  31.54
MB/s
83 pyzmq-16.0.2-p 100% |################################| Time: 0:00:00  16.59
MB/s
84 testpath-0.3-p 100% |################################| Time: 0:00:00   7.07
MB/s
85 tornado-4.5.1- 100% |################################| Time: 0:00:00  20.86
MB/s
86 h5py-2.7.0-np1 100% |################################| Time: 0:00:00  22.93
MB/s
87 html5lib-0.999 100% |################################| Time: 0:00:00   4.87
MB/s
88 jinja2-2.9.6-p 100% |################################| Time: 0:00:00  25.44
MB/s
89 pip-9.0.1-py35 100% |################################| Time: 0:00:00  26.58
MB/s
90 prompt_toolkit 100% |################################| Time: 0:00:00  22.53
MB/s
91 Python-dateuti 100% |################################| Time: 0:00:00  15.33
MB/s
92 qt-5.6.2-vc14_ 100% |################################| Time: 0:00:01  32.13
MB/s
```

```
93  scipy-0.19.0-n 100% |################################| Time: 0:00:00  30.25
    MB/s
94  traitlets-4.3. 100% |################################| Time: 0:00:00   8.55
    MB/s
95  bleach-1.5.0-p 100% |################################| Time: 0:00:00   1.43
    MB/s
96  iPython-6.0.0- 100% |################################| Time: 0:00:00  30.43
    MB/s
97  jupyter_core-4 100% |################################| Time: 0:00:00   0.00
    B/s
98  pyqt-5.6.0-py3 100% |################################| Time: 0:00:00  29.48
    MB/s
99  jupyter_client 100% |################################| Time: 0:00:00  10.31
    MB/s
100 nbformat-4.3.0 100% |################################| Time: 0:00:00   8.85
    MB/s
101 ipykernel-4.6. 100% |################################| Time: 0:00:00   9.57
    MB/s
102 nbconvert-5.1. 100% |################################| Time: 0:00:00  13.14
    MB/s
103 jupyter_consol 100% |################################| Time: 0:00:00   4.86
    MB/s
104 notebook-5.0.0 100% |################################| Time: 0:00:00  29.31
    MB/s
105 qtconsole-4.3. 100% |################################| Time: 0:00:00  11.15
    MB/s
106 widgetsnbexten 100% |################################| Time: 0:00:00  28.75
    MB/s
107 ipywidgets-6.0 100% |################################| Time: 0:00:00   0.00
    B/s
108
109 #
110 # To activate this environment, use:
111 # > activate cntkKeraspy35
112 #
113 # To deactivate this environment, use:
114 # > deactivate cntkKeraspy35
```

```
#
# * for power-users using bash, you must source
#
```

创建完毕之后，激活这个虚拟环境：

```
(d:\Program Files\Anaconda3) E:\code> activate cntkKeraspy35
```

现在可以安装 CNTK 了。如果安装只支持 CPU 的版本，则可以运行如下命令：

```
(cntkKeraspy35) E:\code > pip install https://cntk.ai/PythonWheel/CPU-Only/
cntk-2.0-cp35-cp35m-win_amd64.whl
```

在屏幕上将出现下面所示的信息：

```
Processing https://cntk.ai/PythonWheel/CPU-Only/cntk-2.0-cp35-cp35m-
win_amd64.whl
Requirement already satisfied: numpy>=1.11 in d:\program files\anaconda3\
envs\cntkKeraspy35\lib\site-packages (from cntk==2.0)
Requirement already satisfied: scipy>=0.17 in d:\program files\anaconda3\
envs\cntkKeraspy35\lib\site-packages (from cntk==2.0)
Installing collected packages: cntk
Successfully installed cntk-2.0
```

如果要安装支持 GPU 的版本，则可以运行下面的命令：

```
(cntkKeraspy35) E:\code > pip install https://cntk.ai/PythonWheel/GPU/cntk
-2.0-cp35-cp35m-win_amd64.whl
```

在屏幕上将出现下面所示的信息：

```
Processing https://cntk.ai/PythonWheel/GPU/cntk-2.0-cp35-cp35m-win_amd64.whl
Requirement already satisfied: numpy>=1.11 in d:\program files\anaconda3\
envs\cntkKeraspy35\lib\site-packages (from cntk==2.0)
Requirement already satisfied: scipy>=0.17 in d:\program files\anaconda3\
envs\cntkKeraspy35\lib\site-packages (from cntk==2.0)
Installing collected packages: cntk
Successfully installed cntk-2.0
```

安装好 CNTK 后，就可以安装支持 CNTK 版本的 Keras 了。截至目前，正式版本的 Keras 还没有整合 CNTK，微软研究院正在和 Keras 的作者合作，希望在下一个版本中正式整合 CNTK。目前需要安装微软自己做的服务端仓库克隆（fork）的 Keras：

```
pip install git+https://github.com/souptc/Keras.git
```

在安装完毕以后需要更新 Keras 的配置文件，指定使用 CNTK 作为后台。一般是修改%USERPROFILE%/.Keras/Keras.json 文件为：

```
{
    "epsilon": 1e-07,
    "image_data_format": "channels_last",
    "backend": "cntk",
    "floatx": "float32"
}
```

如果找不到这个文件，则说明 Keras 还没有被启动过，可以手动创建这个文件，并输入以上内容。在 Windows 或者 Linux 中，也可以通过设置系统环境变量 Keras_BACKEND 的值为 cntk 来实现。

在本书接下来的例子中使用的都是基于 GPU 的 CNTK 后台。

我们也将 CNTK 的 Python Wheel 文件以及与 CNTK 配套的 Keras 放到 www.broadview.com.cn 上供读者下载，然后可以通过 pip 的本地安装加载 CNTK 和 Keras。

如果不选择 CNTK，读者也可以在 Theano 和 TensorFlow 中选择一个作为后台计算环境，然后安装 Keras 作为建模环境。

1.2.6 安装 Theano 计算环境

Theano 可以说是基于 Python 的深度学习环境的鼻祖，由蒙特利尔大学的 MILA 研究组开发。Theano 这个名字源于古希腊女数学家 Theano，著名数学家毕达哥拉斯的老婆。

Theano 是一个符号计算环境，对基于 Python 的数学表达式进行编译执行，可以运行于 CPU 或者 GPU 上。很多科研人员都使用 Theano 来开发新的网络结构和算法，也促进了深度学习的普及化，可以说没有 Theano，也许就没有深度学习现在的热潮，至少这个热度会晚来几年。

Theano 的缺点是目前只支持一个 GPU 的运算，不支持多个 GPU 的并行，这造成在大规模的建模训练中 Theano 并不是最好的选择。

通过 GitHub 可以下载最新的 Theano 环境，在下载后的目录中执行 python setup.py install 命令即可将 Theano 安装到当前的 Python 环境中。接下来安装 Keras。以同样的

方法通过 GitHub 克隆最新的 Keras 环境，在下载后的目录中执行 python setup.py install 命令即可安装。

为了让 Theano 有效运行，还需要配置相应的参数。进入“控制面板”，打开“系统”，单击“高级系统设置”，打开“系统属性”对话框，如图1.9所示。单击“环境变量”，输入如下的新环境变量，如图1.10所示。

```
THEANO_FLAGS= floatX = float32,device = gpu,base_compiledir=C:\
Theano_compiledir,[nvcc]compiler_bindir=C:\Program Files (x86)\Microsoft
Visual Studio 12.0\VC\bin
```

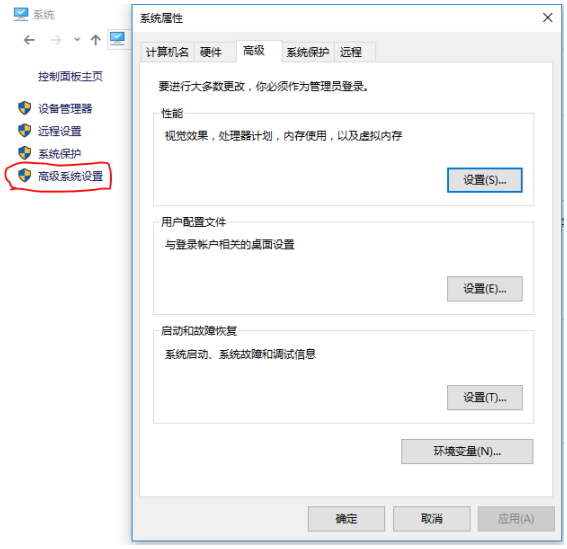


图 1.9 “系统属性”对话框

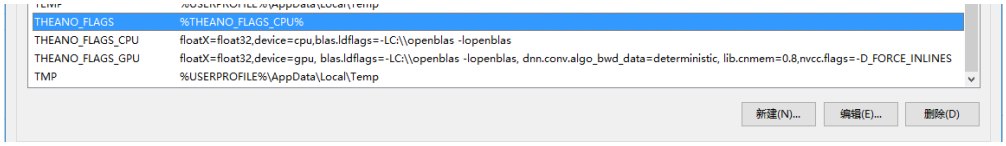


图 1.10 THEANO_FLAGS 的设置

Theano 按照作者的原意是不能与 Python3.5 一起工作的，因此我们需要删除初始化文件__init__.py 中的如下语句：

```
1 if sys.platform == 'win32' and sys.version_info[0:2] == (3, 5):
2       raise RuntimeError( "Theano do not support Python 3.5 on Windows. Use
        Python 2.7 or 3.4.")
```

初始化文件可以在如下目录中找到：

```
%USERPROFILE%\Anaconda3\lib\site-packages\Theano\Theano\
```

为了能给模型绘图，还需要安装两个软件包，即 graphviz 和 pydot。在 Anaconda Console 中输入：

```
conda install graphviz
pip install git+https://github.com/nlhepler/pydot.git
```

如果安装过程有错误，则可以通过以下命令找到相关的环境变量，从而发现出错的原因。

- where Python，这个命令显示 Anaconda 系统安装的目录。
- where gcc, where g++, where gendef, where dlltool，返回 MinGW 的目录。
- where cl，返回 VS 2013 的安装路径。

现在就可以使用 Keras 和 Theano 进行深度学习的模型训练了。

1.2.7 安装 TensorFlow 计算环境

也有一些读者可能倾向于使用 TensorFlow 作为后端计算平台。TensorFlow 是一个很多人所熟知的深度学习计算环境，由 Google Brain 的科学家开发。TensorFlow 开源早，各种教程和培训资料很多，社区非常成熟、活跃，目前是热度排名第一的深度学习计算环境，这是 TensorFlow 最大的优势。背靠 Google 的力量，TensorFlow 在持续开发新的功能上也一步没有落下。TensorFlow 对于多 GPU 和分布式计算的支持是其受到热捧的原因之一。基于 TensorFlow，Google 开发了很多著名的深度学习模型，比如 Inception、Neural Networks for Machine Translation、Generative Adversarial Networks 等。同时基于 TensorFlow 有很多高质量的编程框架方便用户使用。比如本书将要讲解的 Keras 就是其中之一，另外还有 TensorFlow Slim 方便对复杂模型进行定义和建模，特别是图像类模型；PrettyTensor 对于张量类的对象使用链接式（Chainable）语法快速搭建所需模型。

下面我们就来讲解 TensorFlow 的安装。

- 只有 CPU 支持的 TensorFlow。如果电脑没有 NVIDIA 显卡，就必须采用这个版本。这个版本很简单，如果只是“玩票”性质的，我们也推荐使用这个版本。
- GPU 支持的 TensorFlow。TensorFlow 程序在 GPU 上跑得比 CPU 上快数十倍。如果电脑已经安装如下软件/硬件，而且又追求极致的性能，那么推荐使用这个

版本。不过该版本的 TensorFlow 不支持 Windows 操作系统，因此读者需要使用 Docker 来安装。

- CUDA Tookit 8.0，参见前面 CUDA 的安装步骤。同时保证把 CUDA 的路径添加到%PATH% 中。
- NVIDIA 显卡驱动，必须兼容 CUDA Toolkit 8.0。
- cnDNN v5.1. 参见 NVIDIA 文档 https://developer.NVIDIA.com/cudnn。cnDNN 通常不和其他 CUDA DLL 安装在同一个目录下，所以要确保将 cuDNN DLL 的路径添加到%PATH% 里面。
- GPU 显卡必须支持 CUDA Compute Capability 3.0 或者以上版本。参见 https://developer.NVIDIA.com/cuda-gpus 的说明。

TensorFlow 有两种安装方式：pip 安装和 Anaconda 安装。pip 直接把 TensorFlow 安装在本机 OS 下，而不是虚拟环境中；Anaconda 则会把 TensorFlow 安装在虚拟机上。Anaconda 并没有 Google 官方支持，所以 TensorFlow 团队没有测试维护 conda 包，由用户自己承担风险。这里只介绍 pip 安装。

(1) 按照前面的步骤安装 Python 3.5，TensorFlow 只支持 3.5.x 版本。Python 3.5.x 自带了 pip3 包管理工具。

(2) 如果安装支持 CPU 的 TensorFlow 版本，可输入以下命令：

```
> pip3 install --upgrade tensorflow
```

(3) 如果安装支持 GPU 的 TensorFlow 版本，可输入以下命令：

```
> pip3 install --upgrade tensorflow-gpu
```

安装结束以后，可以使用以下步骤验证 TensorFlow 是否已经正确安装。

(1) 启动命令行，输入 Python。

(2) 输入以下程序：

```
import tensorflow as tf
hello = tf.constant('Hello, TensorFlow!')
sess = tf.Session()
print(sess.run(hello))
```

如果安装成功则会输出字符：Hello, TensorFlow!。

至此，TensorFlow 安装成功。

1.2.8 安装 cuDNN 和 CNMeM

如果想要模型训练更加高效，特别是卷积神经网络模型，还需要安装 cuDNN 和 CNMeM 软件包。

cuDNN 是 NVIDIA 开发的专门强化卷积神经网络模型训练的库，全称为 NVIDIA CUDA Deep Neural Network library，支持常见的深度学习软件，比如 CNTK、Caffe、Theano、Keras、TensorFlow 等。cuDNN 对卷积神经网络模型的训练速度能提升 2~3 倍，比如 cuDNN 5.1 在 VGG 模型中相对于不使用该软件包的情况提升大约 2.7 倍。读者可以到 https://developer.NVIDIA.com/cudnn 直接下载安装程序。在下载之前，NVIDIA 会要求注册一个免费的账户。NVIDIA 提供的下载包是一个 ZIP 压缩文件，解压缩后生成一个 cuda 文件夹，里面有三个子文件夹，分别是 bin、include 和 lib。在 bin 文件夹中包含一个 cudnn64_5.dll 文件，在 include 文件夹中包含一个 cudnn.h 头文件，而在 lib 文件夹中包含一个 x64 子目录，内含一个名为 cudnn.lib 的文件。为了能使用 cuDNN，将 cudnn.lib 文件拷贝到 CUDA 安装文件夹的 lib 子目录下，将 cudnn64_5.dll 文件拷贝到 CUDA 安装文件夹的 bin 子目录下，将 cudnn.h 头文件拷贝到 CUDA 安装文件夹的 include 子目录下就行了。

CNMeM 是 NVIDIA 开发的一个显存管理分配软件。预先给深度学习项目分配足够的显存能有效提高训练速度，一般提升 10% 左右。Theano 已经集成了 CNMeM，因此如果需要使用 CNMeM，只需修改.theanorc 配置文件，加入如下语句即可：

```
[lib]
cnmem=0.8
```

cnmem 后面的数值是 GPU 内存分配给 Theano 使用的百分比。这里 0.8 表示 80%。如果这个值设为 0，表示禁止使用 CNMeM 的功能。任何 0~1 之间的实数都行，其对应于百分比。不过实际中会被卡在 95% 左右，剩余的部分供驱动程序使用。如果这个数值大于 1，那么就是指定以 MB 为单位的显存容量，比如 cnmem=50 表示分配 50MB 显存给深度学习使用。

2 数据收集与处理

大数据分析的一个重要组成部分就是数据的收集、存储和组织，特别是相比于传统数据分析，大量非结构化数据的爆炸性增长使得这种需求更加紧迫。在这一章中，介绍几种常见的数据收集、存储、组织以及分析的方法和工具。首先介绍如何构造自己的网络爬虫从网上抓取内容，并将其中按照一定结构组织的信息抽取出来；然后介绍如何使用 ElasticSearch 来有效地存储、组织和查询非结构化数据；最后简要介绍和使用 Spark 对大规模的非结构化数据进行初步分析的方法。

2.1 网络爬虫

网络上充满了大量丰富的信息，通过网络爬虫，可以将相关的信息有组织、有计划地收集起来。这些信息大部分是非结构化的文字、图片或者视频、音频信息。虽然单一的信息碎片所包含的有用信息量不一定高，但是把这些大量的信息系统地收集起来以后，通过合理的综合分析常常能得到意想不到的结果。比如，Click-o-Tron 公司通过分析超过三百万条网上的夸张的新闻标题，运用深度学习技术教会计算机自动生成有吸引力的新闻标题；Narrative Science 公司通过分析大量的商业报表，运用神经网络自动根据数据生成图表以及对图表的解释。现在有了人工智能，以后几乎连初级分析师都不需要了！

因此，本节将要介绍如何使用 Python 构造自己的网络爬虫。

2.1.1 网络爬虫技术

当浏览网页的时候，我们会在浏览器中输入想要显示的网页地址，然后浏览器会将相应网页上的信息取回来，并显示在浏览器中。我们的眼睛会扫描显示的内容，并选择自己想要的部分，比如文字或者视频信息，而忽略不想要的信息，比如广告等。这一切都可以通过执行一套程序来自动完成，这套程序通常就被称为“网络爬虫”。

正如上面所提到的，任何网络爬虫程序都是将我们浏览网页的行为自动化、程序化，因此一般都遵照如下步骤进行。

(1) 准备需要访问的网页完整地址，即网页的域名加上查询字符串，比如要在 Bing 上搜索 python scarpy，其完整的网址是：http://www.bing.com/search?q=python+scrapy。在这个地址中，http://www.bing.com/search 是网页域名，而问号后面的就是查询字符串：q=python+scrapy，表明查询的内容是与 Python 或者 Scrapy 相关的网页。

(2) 得到所要访问的网页地址以后，还需要确定访问方式。一般来说，访问网页有两种方式，即 GET 或者 POST。顾名思义，GET 是直接将网页上的数据取回来，而 POST 是往指定网页上注入数据，比如在登录页面中填写的 ID 和密码等。有时还需要填写头部信息，即 Header，以及指定 Cookie。有些网站要求必须启用 Cookie 才能正常访问。

(3) 提交了网页请求后，即可获得请求的响应，即 Response，通常会得到这个网页的源码。

(4) 得到网页源码之后并不能直接使用，还需要对信息进行分解，从结构化源码中提取所需要的内容，这个过程称为 Parsing。因为一般返回的网页源码都是 HTML 的，可以对源码进行解析，从而提取内容。一般使用现成的软件包就可轻松实现这一过程。在 Python 中，比较有名的是 BeautifulSoup。

在 Python 中有很多不同的网络爬虫框架在互相竞争。本书选择介绍一种非常强大的工具——Scrapy。Scrapy 是使用比较广泛的一种数据抽取工具，不仅可以用来对网站内容进行爬取，并抽取结构化数据，还可以通过不同的 API 来抽取特定网站的特定数据，比如亚马逊 Associate Web Service 就可以通过 Scrapy 接入其 API 来下载内容。因此，Scrapy 是一个功能强大的通用的数据收集工具。

本节将通过案例方式来介绍如何构造和使用 Scrapy 来爬取网站并抽取相应的内容。读者可以通过修改本节的例子来得到合乎自己需要的爬虫。如果你没有安装 Scrapy，则可以通过以下方式来安装。

(1) 如果已经安装了 Anaconda 发布的 Python，则可以直接在命令行中输入如下命令来安装 Scrapy：

```
conda install -c scrapinghub scrapy
```

(2) 如果 Python 不是 Anaconda 发布的，那么可以通过 pip 来安装：

```
pip install Scrapy
```

在安装好 Scrapy 以后，就可以根据下面的步骤构造自己的网络爬虫了。

2.1.2 构造自己的 Scrapy 爬虫

构造一个 Scrapy 爬虫非常简单，基本遵循以下几步就可以完成。

(1) 在一个空白的目录中创建一个 Scrapy 项目，通过执行如下命令来实现，如图2.1所示。

```
scrapy startproject project_name
```

其中，project_name 是要创建的项目名称，可以是任何合理的字符组合。



图 2.1 生成 Scrapy 项目

(2) 定义要抽取的内容，通过项目目录中的 items.py 文件来实现。

(3) 定义爬虫的目标网站和爬虫的具体行为，通过在 spider 子目录中定义一个基于 Python 的爬虫程序（通常叫做 spider.py）来实现。

(4) 定义对抽取出来的数据的操作，比如是存为文本文件还是存入某种数据库中等，通过项目目录中的 pipelines.py 文件来实现。

(5) 定义爬虫的一般设定，通过项目目录中的 settings.py 文件来实现。

下面通过爬取网易财经新闻的例子来详细介绍以上每一步的具体实现。选择爬取网易财经新闻，是因为网易的新闻网页地址规律，网页定义规范，同时收集的信息也是比较有意思的。举例来讲，网易财经新闻的一个典型网址是：http://money.163.com/17/0301/10/CEEF06PH002581PQ.html，其中，基本网址是 http://money.163.com，这是所有网易财经新闻都有的；在后面的/17/0301/10/CEEF06PH002581PQ.html 部分，我们发现前 6 个数字/12/0301 对应的是“/年/月日”的结构，后面的 10 含义不明，猜测是 24 小时计的新闻发布时间，因为所有新闻链接这里都是一个两位数字，在 1~24 之间；最后的字符串 CEEF06PH002581PQ 应该是新闻页面的 ID。

首先创建一个空白的目录，比如叫 money163crawler，在 Windows 系统的命令行中输入如下命令：

```
mkdir money163crawler
```

进入这个空白的目录，创建自己的 Scrapy 项目，并为这个网易财经新闻爬虫项目取名为“money163”。在命令行中输入如下命令：

```
scrapy startproject money163
```

这时，Scrapy 自动在 money163crawler 目录下生成一个新的目录 money163，其中又包含一个同名的子目录和一个 scrapy.cfg 配置文件。在这个同名的子目录下包含一系列 Python 文件，如__init__.py、items.py、settings.py、pipelines.py 等，同时还有一个子目录 spiders，在该子目录下包含一个__init__.py 初始化文件。这两个__init__.py 初始化文件一般都是空白的，我们暂时不需要去理会它们，将主要精力集中在 items.py、settings.py、pipelines.py 和将要在 spiders 子目录下生成的爬虫程序（比如可以叫作 money_spider.py）上。

现在，Scrapy 项目的基本结构已经建立起来了，如图2.2所示。

```
money163\
        scrapy.cfg                      #配置文件
        money163\                       #项目的Python模块，代码导入入口
                __init__.py
                items.py                #定义抓取内容
                pipelines.py            #定义最终数据处理
                settings.py             #项目设置文件
                spiders\                #爬虫程序所在目录
                        __init__.py
                        money_spider.py
```

图 2.2　Scrapy 项目结构示意图

基本结构建立起来以后，我们需要按照上面的步骤一次完成对内容抽取，爬虫目标和行为以及数据操作的定义，每一个定义都对应一个文件，下面依次介绍。

首先，在 items.py 程序中定义需要抽取的内容，这基本上是通过定义一个继承于 scrapy.Item 的内容类来完成的。每一个内容都属于 scrapy.Field()，定义非常简单，即内容名称 = scrapy.Field()。比如对于新闻，通常需要定义新闻题目、新闻链接、新闻主体等。下面代码展示了如何在自己的类中定义这三个内容。

```
1  class MoneyNewsItem(scray.Item):
2          #define as name = scrapy.Field()
3          news_title = scrapy.Field()
```

```
4           news_body = scrapy.Field()
5           news_url = scrapy.Field()
```

如果需要抽取更多的内容，按照上面的方式继续定义就行。至此，对于抽取内容的定义就完成了。

其次，我们先要定义爬虫的起始网站和具体行为。这需要在 spiders 子目录下新建立一个 Python 文件，可以命名为“money_spider.py”，因为爬取的是网易财经网页，统一子域名是 money。这个文件比较复杂，可以继承不同的类来定义，在这里我们使用 Scrapy 的 CrawlSpider 类。在这个文件中，需要定义三个主要内容：一是爬虫的名字；二是目标网站，包括爬取模式和对返回链接的过滤等；三是从返回的对象中按照其结构抽取所需要的数据。

爬虫名字的定义最简单：name = "myspider"，即定义了爬虫的名字为“myspider”。

接下来还要定义爬虫的目标网站或者说起始网站，通过 start_urls=[...] 即可实现，在列表里可以定义要从哪些具体的网页开始爬取，可以是一个网页，也可以包含多个网页，比如：

```
start_urls = ['http://money.163.com/', 'http://money.163.com/stock/']
```

表示依次从网易财经主页和网易财经股票主页开始爬取。对具体的网页进行爬取时，爬虫会把该网页上的所有元素都抓取下来，包括里面包含的其他链接。这时候我们需要告诉爬虫如何处理这些链接，比如是否按照这些被抓取的链接指向继续不停地爬取相应的网页；对于这些返回的链接是否需要过滤，如果要过滤，过滤的规则是什么……这些都可以通过 rules 变量来定义。下面是一个典型的爬取网页的规则定义。

```
1  rules = Rule(
2                LinkExtractor(allow=r"/\d+/\d+/\d+/*"),
3                follow=True,
4                callback="moneyparser"
5  )
```

在这个规则里，首先，通过 LinkExtractor 定义了对返回的链接的过滤条件，这里通过正则表达式/\d+/\d+/\d+/* 来定义，这个正则表达式的含义很明确，就是返回的链接如果是以“/数字/数字/数字/任意字符”形式出现的就接受，否则过滤掉。

其次，follow=True 告诉爬虫，对返回的链接持续递归地进行抓取。

最后，callback="moneyparser" 指定的是一个函数名，告诉爬虫对于返回的 response 对象，下载后使用该函数进行处理。这个函数通常就是对返回对象的数据进行实际的抽取动作。在详细介绍 moneyparser 函数之前，我们来总结一下定义目标网站的语句。

```
allowed_domains=["money.163.com"]
start_urls=['http://money.163.com/', 'http://money.163.com/stock/']
rules =Rule(
                LinkExtractor(allow=r"/\d+/\d+/\d+/*"),
                follow=True,
                callback="moneyparser"
        )
```

总的来说，在 Scrapy 中可以非常方便地制定爬虫的规则。下面我们来看看 callback 里面提到的 moneyparser 函数。

moneyparser 函数对返回的 response 对象使用 xpath 进行解析，抽取所需要的具体数据。

```
def moneyparser(self,response):
        item = MoneyNewsItem()
        title=response.xpath("/html/head/title/text()").extract()
        if title:
           item['news_title']=title[0][:-5]

        news_url=response.url
        if news_url:
           item['news_url']=news_url

        news_body=response.xpath("//div[@id='endText']/p/text()").extract()
        if news_body:
           item['news_body']=news_body
```

是不是也很简单？在这个函数里，对新闻标题和正文使用 xpath 进行了抽取（extract），如果要抽取其他一些要素，可以参考 Scrapy 的教程，这里就不一一解释了。但是新闻链接是不用抓取的，直接在 response 对象里存有。

规则和方法都定义好了以后，这个爬虫就可以从一个给定的主页，比如 money.163.com 抓取所有内容了，在所抓取的内容中按照规定的方法抽取定义好的具体要素；同时从内容中找出超链接，如果符合规则就继续下载该超链接对应的页面。

接下来，我们需要对所抽取的具体要素进行处理，要么显示在终端的窗口中，要么存入某个地方或者数据库中。作为演示，我们将抽取出来的要素构造成一个词典，以 JSON 文档的格式存为文本文件，每个页面单独存成一个文件。这个过程可以很容易地在 pipelines.py 程序里定义。这个过程需要定义一个类，这个类里只有一个方法——

process_item(self, item, spider)。因为在返回的 item 要素列表里面含有超链接地址，而这个地址的最后部分就是新闻页面的 ID，因此我们可以使用这个 ID 作为文件名，这是一个很容易实现的过程。

```
class MyPipeline(object):
    def process_item(self, item, spider):
          url = item['news_url']
          filename = url.split("/")[-1].split(".")[0]
          fo = open(filename, "w", encoding="UTF-8")
          fo.write(str( dict(item) ))
          fo.close()
          return None
```

为了使爬虫能正确运行，最后还需要进行如下设置。

- BOT_NAME：这是爬虫的名字，当在项目目录中用命令行执行操作时，Scrapy 知道调用哪个爬虫；这个名字也被会用来构造 User-Agent 和记录日志。
- SPIDER_MODULES：定义了具体的爬虫，Scrapy 会根据这个列表中的内容来找爬虫。
- NEWSPIDER_MODULE：这个选项指定使用 genspider 命令来新生成爬虫的路径。
- ITEM_PIPELINES：这是一个列表，用于指定执行顺序。在每一个需要执行的 pipeline 中都用一个数字定义了其顺序，数字越小，执行的优先级越高。

下面给出本章所讲例子的设置，供参考。

```
BOT_NAME = 'money163'
SPIDER_MODULES = ['money163.spiders']
NEWSPIDER_MODULE = 'money163.spiders'
ITEM_PIPELINES = {'money163.pipelines.MyPipeline':300,}
```

至此，爬虫就建立完毕，可以用来爬取网页了。但是这个简单的爬虫只能爬取固定的起始网页，在实际应用中作用有限。下面我们将对这个简单的爬虫进行功能上的拓展，主要是让这个爬虫能接受不同的参数来改变爬取网页的行为。比如，可以指定爬虫爬取某一天的网易财经网页，这样就可以累积各种财经新闻供日后分析所用了。

2.1.3 构造可接受参数的 Scrapy 爬虫

这一节介绍如何改造上面的简单的爬虫，使其能接受一些参数，这些参数能改变起始网页的地址，从而使用同一个爬虫爬取不同的网站。我们知道，网易不仅财经新闻符合前面提到的页面链接模式，即页面以“money.163.com+\年\月日\数字\任意字符”形式出现，这一规律也适用于其他网易子站，比如网易科技等。因此，只要能改变起始网页链接地址，就能爬取不同网站的页面了。

做到这一点非常简单，只需在爬虫程序里新建立一个初始化方法，该方法接受参数，然后将 start_urls 放进去，修改为因这个参数而变化，这个功能就实现了。示例如下：

```
class ExampleSpider(CrawlSpider):
    name = "stocknews"

    def __init__(self, site='money.163.com', *args, **kwargs):
        allowrule = r"/\d+/\d+/\d+/*"
        self.counter = 0
        self.stock_id = id
        self.allowed_domain=[site]
        self.start_urls = ['http://\%s' \% (site)]
        super(ExampleSpider, self).__init__(*args, **kwargs)
```

在这段代码里，根据所提供的起始网页不仅修改了 start_urls，同时也修改了 allowed_domain，保证爬虫顺利进行。最后通过 super 方法执行这个类来更新数据。

这么修改以后，在运行时需要给爬虫提供参数，这与在命令行执行方法和借用别的程序调用时略有差异，下一节详细解释。

假如现在不仅需要灵活定义爬取的网站，还需要定义抓取哪一天的新闻，那么不仅需要修改 start_urls 这个变量，还需要修改爬取规则 rules 一项。有了上面的例子，这个修改也是很容易的。按照前面的定义，修改爬取网页的规则，主要是对 LinkExtractor 里面所允许的规则进行修改，使其按照输入参数进行改变，这可以通过下面两句来实现：

```
allowrule = "/\%s/\%s\%s/\d+/*" \% (year, month, day)
ExampleSpider.rules=(Rule(LinkExtractor(allow=allowrule), callback="
parse_news", follow=True),)
```

我们看到，修改 allowrule 变量的方法和前面一样，但不是使用 self.rules 来定义新的规则，而是必须要引用所在类的名字，即 ExampleSpider。

通过引入参数，我们可以设计一个比较灵活的爬虫来爬取多种类型的网站或者不同时间、不同子站的网页。下面将要介绍如何运行 Scrapy 爬虫。

2.1.4 运行 Scrapy 爬虫

运行 Scrapy 爬虫有两种方法：一种是在命令行里执行 crawl 指令；另一种就是在别的程序中调用 Scrapy 爬虫。

在命令行里运行 Scrapy 爬虫非常简单，进入项目的主目录，即包含 scrapy.cfg 文件的那个目录中，输入：

```
scrapy crawl money163
```

即可让爬虫按照爬虫程序里定义好的规则和方法爬取网页。这里 money163 是在 spider.py 程序文件中使用 name = "money163" 定义的爬虫名字。crawl 是让 Scrapy 爬虫开始爬取网页。

如果爬虫可以接受不同的参数，比如上面例子中使用不同的起始网址，那么需要使用-a parameter = value 的方式提供参数值。比如要求从 money.163.com/stock 开始爬取网页：

```
scrapy craw money163 -a site = money.163.com/stock
```

需要留意的是，所有参数都是以字符串形式传递给爬虫的，即使参数需要输入数字，比如-a year=2017，也不需要使用引号。

在别的程序中调用 Scrapy 爬虫比在命令行中调用稍微复杂一点，但是也比较直观。下面以在 Python 程序里调用刚才写好的 money163 爬虫为例来介绍这个方法。一般来说，在别的程序里调用 Scrapy 爬虫可以使用不同的类。这里使用 CrawlerProcess 类，配合 get_project_settings 方法，就可以在项目目录中非常方便地使用别的程序运行自己的爬虫了。

首先引入相应的模块和函数：

```
1 from scrapy.crawler import CrawlerProcess
2 from scrapy.utils.project import get_project_settings
```

然后定义爬虫过程。在定义的过程中，先通过 get_project_settings 获取项目的信息，再传给所定义的爬虫过程：

```
process = CrawlerProcess(get_project_settings())
```

定义好爬虫过程后，只需调用这个过程对象，包括传递参数，就能运行爬虫了。比如：

```
process.crawl('stocknews', id=id, page=page)
```

下面的示例代码展示了在 Python 里面如何调用上面建立的爬虫，包括应用不同的参数。

```
from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings

process = CrawlerProcess(get_project_settings())

# 'stocknews' is the name of one of the spiders of the project.
for site in ['money.163.com', 'tech.163.com', 'money.163.com/stock']:
        process.crawl('myspider', site=site)
process.start() # the script will block here until the crawling is finished
```

在这个程序中，通过 process.crawl()，按照列表中的三个网址定义了三个爬虫，最后通过 process.start() 来启动爬虫。需要注意的是，因为使用了 get_project_settings，这个 Python 程序需要在项目所在目录下执行才能有效运行。

另外，需要注意的是，从命令行执行和从程序内部调用 API 在并发运行爬虫的数量上会有差别。当通过命令行 scrapy crawl 运行爬虫时，默认方法是一个线程处理一个爬虫；而通过 Python 程序调用 API 的方式则默认为同时调用多个爬虫。比如上面的程序要求将列表中的三个网址作为起始网址，则会同时启动三个爬虫下载网页。

同时启动多个爬虫虽然会充分利用 CPU 和带宽，但是在有些情况下，我们希望顺序执行每个爬虫，这时调用的程序会有一点小的改变，需要使用 twisted 包里的 internet.defer 方法将每个爬虫串联起来，同时调用 reactor 来控制执行的顺序。在下面的例子中，首先定义一个函数，里面通过 yield process.crawl('myspider', site=site) 将爬虫串联起来，但是不执行。最后调用 reactor.run() 来顺序执行爬虫。

```
from twisted.internet import reactor, defer
from scrapy.crawler import CrawlerProcess
from scrapy.utils.project import get_project_settings

process = CrawlerProcess(get_project_settings())

sitelist = ['money.163.com', 'tech.163.com', 'money.163.com/stock']
```

```
@defer.inlineCallbacks
def crawl(sitelist):
      for site in sitelist:
           yield process.crawl('myspider', site=site)
      reactor.stop()

crawl()
reactor.run()
```

Scrapy 也可以在多台机器上部署分布式的爬虫，限于篇幅，这里就不详细介绍了，有兴趣的读者可以参考 Scrapy 的手册和帮助文档。

2.1.5 运行 Scrapy 爬虫的一些要点

爬虫在运行时会对目标网站有一定的压力，特别是同时并发运行很多爬虫的时候。另外，很多网站会对网络请求是否是网络爬虫进行识别，如果发现是网络爬虫，则会进行约束，比如限制流量甚至直接拒绝响应。因此，在构造自己的爬虫时需要有这方面的考虑，主要通过合理设置 settings.py 文件里面的选项来实现。

(1) 通过不停地替换不同的 User-Agent 来降低被网站识别出来的概率。User-Agent 是用户向服务器表明自己身份的字符串，Scrapy 爬虫默认的 User-Agent 是 Scrapy/VERSION (+http://scrapy.org) 。通过将其设置为网络常见的 User-Agent 字符串，甚至设置为一个包含多个常见的 User-Agent 字符串的列表，Scrapy 爬虫在很多不具备精密探测算法的服务器面前就伪装得像普通的常见的网络请求一样，有效降低了被服务器屏蔽的概率。读者可以很容易地通过搜索引擎找到这些常见的 User-Agent 字符串。

当把 User-Agent 选项设置为包含多个常见的字符串的列表时，还需要构建一个 middleware.py 程序来随机调用列表中的某一个字符串提供给服务器表明身份。当然，在使用这个功能之前，需要先在 settings.py 设置程序中做两件事。

一是配置好多个 User-Agent 的列表：

```
USER_AGENTS = [
    "Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1; SV1; AcooBrowser; .
    NET CLR 1.1.4322; .NET CLR 2.0.50727)",
    "Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 6.0; Acoo Browser; SLCC1;
     .NET CLR 2.0.50727; Media Center PC 5.0; .NET CLR 3.0.04506)",
    "Mozilla/4.0 (compatible; MSIE 7.0; AOL 9.5; AOLBuild 4337.35; Windows
    NT 5.1; .NET CLR 1.1.4322; .NET CLR 2.0.50727)",
```

```
    "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.20 (
    KHTML, like Gecko) Chrome/19.0.1036.7 Safari/535.20",
]
```

二是指明 DOWNLOADER_MIDDLEWARES 是哪个文件。默认值是没有这个中间件的，现在有了，比如叫 RandomUserAgent，那么这个值就要设置为一个词典：

```
DOWNLOADER_MIDDLEWARES = {
    'money163.middleware.RandomUserAgent': 1,
}
```

其数值表明执行的顺序，值越小越早执行。

下面的 middleware.py 程序例子就展示了如何随机调用列表中的字符串。

```
import random

class RandomUserAgent(object):
    """Randomly rotate user agents based on a list of predefined ones"""

    def __init__(self, agents):
        self.agents = agents

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.settings.getlist('USER_AGENTS'))

    def process_request(self, request, spider):
        request.headers.setdefault('User-Agent', random.choice(self.agents))
```

这个程序做了两件事：一是从爬虫的设置文件中获取 USER_AGENTS 项的值；二是将从列表中随机选中的 User-Agent 注入当前的爬虫请求中。

(2) 通过合理设置以下选项，可以有效降低被服务器屏蔽的概率。

- DOWNLOAD_DELAY——这个设置控制的是下载器在连续下载网页之间所需要等待的时间，默认值为 0，单位为秒。该设置可以控制爬虫的速度，防止目标服务器过载。如果设置的数值含有小数位，则对应的单位是毫秒，比如 DOWNLOAD_DELAY = 2.05 表示等待时间是 2 秒又 50 毫秒。
- DOWNLOAD_TIMEOUT——这是下载器在超时之前需要等待的秒数，默认值为 180 秒。设置为较小的数值较好。

- CONCURRENT_REQUESTS——这是 Scrapy 下载器能同时发送的请求数量，默认值为 16。可以设置为较小的值来控制流量。
- CONCURRENT_REQUESTS_PER_DOMAIN——这是 Scrapy 下载器同时发送到一个单一域名的请求数量，默认值为 8。
- CONCURRENT_REQUESTS_PER_IP——这是限制 Scrapy 下载器同时发送到某一个 IP 地址的请求数量。默认值为 0，表示无限制。如果这个值不为 0，则该设置有高于 CONCURRENT_REQUESTS_PER_DOMAIN 的优先级，限制数量以 IP 地址为单位，而不是域名。同时该设置影响 DOWNLOAD_DELAY，现在等待时间也是以 IP 地址为单位的。
- COOKIES_ENABLED——是否启用 Cookie 中间件。默认为启用，但是 Cookie 可以用于用户识别，因此通过连续跟踪某一用户的行为可以被有些网站用来识别是否是网络爬虫。禁止这个中间件能减少被识别出来的概率。当然，有些网站规定必须使用 Cookie，这时候只有通过别的方法来降低被识别出来的概率了。

2.2 大规模非结构化数据的存储和分析

根据维基百科的定义，非结构化数据是指没有定义结构的数据。一种典型的非结构化数据是文本，包括日期、数字、人名、事件等。这样的数据没有规则可寻，所以很难用传统的手段和存在于关系型数据库里的数据做比较。

处理非结构化数据的技术，比如数据挖掘、自然语言处理（NLP）、文本分析等提供了不同的方法从非结构化数据里找出模式。处理文本常用的技巧通常涉及用元数据或者词性标签手动标记。

非结构化数据还包括书籍、杂志、文档、元数据、声音、视频、模拟数据、图像和非结构化文本，比如网页、笔记等。有些数据虽然封装在特定的结构里，但是它们仍然被称为非结构化数据，比如 HTML 文件，虽然 XML 标记形成了树状结构，但是这些标记仅用于渲染，并不代表数据的含义或解释，所以它们依然是非结构化数据。

简单而言，非结构化数据是不能存储在传统的关系型数据库里的数据。

结构化数据是有组织的数据，比如关系型数据库里的信息，这也是谷歌内部对结构化数据的一个概括。当信息高度结构化和可预测时，就可以很容易地以创造性的方式组织和显示。对于前面提到的非结构化数据，通过结构化数据标记来进行组织可以实现一定程度的结构化。结构化数据标记是一种基于文本的数据组织形式，可以在本地文件中存储，也可以以网络服务的方式提供给第三方。

结构化数据标记一般用 JSON-LD 格式表达，下面是一个具体例子。

```
<script type="application/ld+json">
{
  "@context": "http://schema.org",
  "@type": "Message",
  "Subject": "This is a test message",
  "Attachments": [{
    "@type": "Attachment",
    "Name": "Attachment 1",
    "Size": "20k",
    "Body": "abcedfg",
  }]
}
</script>
```

结构化数据标记描述了非结构化数据本身的内容和属性等元数据。比如一个网站有各个品牌的衣服，就需要用标记语言表述每个品牌的属性，比如品牌风格、使用人群、价格区间、用户评价等。当一个网站的所有页面都包含了结构化数据标记时，那么搜索引擎爬取这个网站时，就会把结构化数据应用到两种场景中。

(1) 搜索结果。结构化数据如品牌服装、使用人群、用户评价等会出现在搜索结果中。

(2) 知识图谱。对网站上的内容如果网站的作者是最终结论者，那么搜索引擎可以把这个内容视为事实导入知识图谱中，从而在搜索结果中提供显著的答案。知识图谱代表了有关组织和时间的事实性数据，比如诺贝尔奖官网上的数据。当搜索关键词和诺贝尔奖有关时，在搜索结果中就会返回诺贝尔奖官网的显著链接。

对非结构化数据进行组织时，一般使用 schema.org 定义的类型和属性作为标记（比如 JSON-LD），而且这个标记要公开。比如在谷歌的搜索引擎里面会标记所有有关的网页，而且有标记的页面不向搜索引擎隐藏。

当单个网页上有多种实体类型时，这些实体应该都被标记。举例来讲：

- 品牌服装页面包含品牌介绍和代表视频，应分别使用 schema.org/clothes 和 schema.org/VideoObject 来标记这些类型。
- 列出几种不同品牌的类别页面，应使用相关的 schema.org 类型来标记每个实体，例如产品类别页面的 schema.org/Brand。
- 视频播放页面可能会将相关视频嵌入页面中的单独部分。在这种情况下，标记主要视频以及相关视频。

对于图像数据，也有一些标定的规定，比如将图像 URL 标记为类型的属性时，请确保该图像实际上属于该类型的实例。比如把 schema.org/image 标记为 schema.org/NewsArticle 的属性，则标记图像必须直接属于该新闻文章。所有图片网址都应该可爬取和可索引；否则，搜索引擎无法在搜索结果页面中显示。

2.2.1 ElasticSearch 介绍

ElasticSearch 是一个使用 Java 开发的开源的企业级搜索引擎，当前非常流行。其基于 Lucene 的搜索服务器，提供分布式全文搜索引擎，基于 RESTful Web 接口，具备多用户能力。ElasticSearch 的特点如下：

- ElasticSearch 是一个分布式支持 REST API 的搜索引擎。每个索引都使用可分配数量的完全分片，每个分片可以用多个副本。该搜索引擎可以在任何副本上操作。
- 多集群、多种类型，支持多个索引，每个索引支持多种类型。索引级配置（分片数、索引存储等）。
- 支持多种 API，比如 HTTP RESTful API、Native Java API，所有 API 都执行自动节点操作重新路由。
- 面向文件，不需要前期模式定义，可以为每种类型定义模式以定制索引过程。
- 可靠，支持长期持续性地异步写入。
- 近实时搜索。
- 建立在 Lucene 之上，每个分片都是一个功能齐全的 Lucene 索引，Lucene 的所有权利都可以通过简单的配置/插件轻松暴露。
- 操作具备高度一致性，单个文档级操作是原子的、一致的、隔离的和耐用的。
- 开源许可友好，使用的是 Apache 许可证下的开放源码版本 2（“ALv2”）。

下面介绍 ElasticSearch 的下载、安装、使用和配置。

ElasticSearch 支持很多操作系统，这里介绍安装在 Windows 系统下。如果想安装在其他系统下，请确认 ElasticSearch 的支持，具体列表在这里：https://www.elastic.co/support/matrix。

ElasticSearch 是用 Java 实现的，要求在 Java 8 虚拟环境中运行。ElasticSearch 官网推荐 1.8.0_73 或者更高版本。如果使用了不兼容版本，ElasticSearch 会启动失败。

关于对 ElasticSearch 版本的选择：为了让 ElasticSearch 支持中文分词插件，我们不建议安装 ElasticSearch 标准版本，而是安装 RTF 版本，该版本已经做好相关配置，可以让使用者节省很多时间。

安装 ElasticSearch RTF（Windows）执行以下步骤。

(1) 保证机器上已经安装了 Java 8 虚拟机，如果没有安装，请去 Oracle 官网下载合适的操作系统版本。

(2) 下载 ElasticSearch RTF 版本：

```
git clone git://github.com/medcl/elasticsearch-rtf.git -b master --depth 1
```

(3) 运行：

```
cd Elasticsearch/bin
elasticsearch.bat
```

在命令行窗口中会显示：

```
[2017-04-20T23:42:05,385][INFO ][o.e.h.HttpServer         ] [qBJbdnQ]
publish_address {127.0.0.1:9200}, bound_addresses {127.0.0.1:9200},
{[::1]:9200}
[2017-04-20T23:42:05,386][INFO ][o.e.n.Node               ] [qBJbdnQ]
started
```

然后在浏览器中访问 http://localhost:9200，如果显示：

```
{
     name: "qBJbdnQ",
     cluster_name: "elasticsearch",
     cluster_uuid: "CHZeVrVRSPqmI20I2XEJyQ",
     version:
    {
    number: "5.1.1",
    build_hash: "5395e21",
    build_date: "2016-12-06T12:36:15.409Z",
    build_snapshot: false,
          lucene_version: "6.3.0"
     },
     tagline: "You Know, for Search"
}
```

则说明 ElasticsSearch 启动成功。

2.2.2 ElasticSearch 应用实例

接下来我们用一个实际案例来说明如何应用 ElasticSearch。

成都市政府在网上公开了所有政府公文，我们使用前面介绍的爬虫，自己写个程序将这些文件都抓取下来，这里的细节限于篇幅就不介绍了，有兴趣的读者可以自己写一个。这个例子会把这些文件保存到 ElasticSearch 中，然后提供基于机器学习的文本摘要和搜索功能。

首先建立索引，索引名字为“chengdugov”，发送如下请求：

```
PUT   http://localhost:9200/chengdugov
```

收到响应：

```
HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 48

{"acknowledged":true,"shards_acknowledged":true}
```

表明索引建立成功。

然后建立映射。保存在 ElasticSearch 中的文件是包含结构化数据标记的，所以需要为这个索引建立若干映射。

先看一个文件例子，这个 JSON 格式的文件已经预先处理了，这些结构化数据标记是通过解析网页文本获取的。

```
{
    "题目": "成都市幼儿园管理办法",
    "内容": "第一章　总　则 第一条　（目的依据） 为规范幼儿园管理，促进学前教育事业健康发展，根据《中华人民共和国教育法》、《中华人民共和国民办教育促进法》和国务院《幼儿园管理条例》等法律、法规，结合成都市实际，制定本办法。…… [编者注：文件内容太长，此处省略] …… 第三十七条　（对违反配套幼儿园建设移交规定的责任追究） 违反本办法规定，擅自改变规划配套建设幼儿园用途的，由有关部门依据职权责令改正，并依法追究责任；逾期不移交的，由建设行政管理部门责令限期改正，作为不良信用记录计入成都市房地产开发企业信用信息管理系统，并予以公示。 第六章　附　则 第三十八条　（术语定义） 本办法所称幼儿园，是指对三周岁以上学龄前儿童实施保育和教育的学前教育机构。
```

```
    公益性幼儿园，是指经区（市）县教育行政主管部门认定，执行政府定价，接受财
    政补助的幼儿园。 第三十九条 （施行日期） 本办法自2014年3月1日起施行。",
    "填报时间": "2014-01-30",
    "责任单位": "市政府办公厅",
    "文　　号": "政府令第183号",
    "签发单位": "",
    "签发时间": "2014-01-21",
    "生效时间": "2014-01-21"
}
```

这个文件有 8 个属性：题目、内容、填报时间、责任单位、文号、签发单位、签发时间、生效时间。需要为这些属性建立映射，可以通过发送如下网络请求来实现：

```
POST http://localhost:9200/gov/_mapping/Fulltext
{
"properties": {
            "题目": {
                "type": "text",
                "analyzer": "ik_max_word",
                "search_analyzer": "ik_max_word",
                "include_in_all": "true",
                "boost": 8
            },
            "内容": {
                "type": "text",
                "analyzer": "ik_max_word",
                "search_analyzer": "ik_max_word",
                "include_in_all": "true",
                "boost": 8
            },
              "责任单位": {
                "type": "text",
                "analyzer": "ik_max_word",
                "search_analyzer": "ik_max_word",
                "include_in_all": "true",
                "boost": 1
            },
              "签发单位": {
```

```
                    "type": "text",
                    "analyzer": "ik_max_word",
                    "search_analyzer": "ik_max_word",
                    "include_in_all": "true",
                    "boost": 1
                },
                  "文    号": {
                    "type": "text",
                    "analyzer": "ik_max_word",
                    "search_analyzer": "ik_max_word",
                    "include_in_all": "true",
                    "boost": 1
                },
                  "填报时间": {
                    "type": "date",
                    "format" : "YYYY-MM-dd",
                    "boost": 1
                },
                  "签发时间": {
                    "type": "date",
                    "format" : "YYYY-MM-dd",
                    "boost": 1
                },
                  "生效时间": {
                    "type": "date",
                    "format" : "YYYY-MM-dd",
                    "boost": 1
                }
            }
    }
```

如果映射建立成功的话，网络会响应：

```
HTTP/1.1 200 OK
content-type: application/json; charset=UTF-8
content-length: 21

{"acknowledged":true}
```

接下来可以插入数据了。这里假设我们已经通过前面学习的网络爬虫把文件从成都市政府官网上爬下来并整理成了带结构化标记的数据。URL 结尾是文档 ID，不要重复使用 ID，不然会覆盖就文档。我们使用如下命令来插入数据：

```
POST http://localhost:9200/chengdugov/fulltext/1

{
    "题目": "成都市历史建筑保护办法",
    "内容": "第一条　（目的依据） 为加强对历史建筑的保护，继承和弘扬优秀历史文化，促进城乡建设与社会文化协调发展，根据国务院《历史文化名城名镇名村保护条例》等法律法规，结合成都市实际，制定本办法。…… [编者注： 内容过长，中间省略]……第三十条　（责任追究） 行政机关及其工作人员在历史建筑保护管理中不按照本规定履行职责，玩忽职守、滥用职权、徇私舞弊的，依法给予行政处分。构成犯罪的，依法追究刑事责任。 第三十一条　（施行日期） 本办法自2014年12月1日起施行。",
    "填报时间": "2014-11-12",
    "责任单位": "市政府办公厅",
    "文　　号": "政府令第186号",
    "签发单位": "",
    "签发时间": "2014-10-17",
    "生效时间": "2014-10-17"
}


POST  http://localhost:9200/index/fulltext/2
{
    "题目": "成都市规范行政执法自由裁量权实施办法",
    "内容": "第一条　（目的依据） 为规范行政执法自由裁量权，促进合法行政、合理行政，维护公民、法人和其他组织的合法权益，根据有关法律、法规及《四川省规范行政执法裁量权规定》，结合成都市实际，制定本办法。……[编者注： 内容过长，中间省略]……第二十七条　（施行日期） 本办法自2014年11月1日起施行。2010年6月24日成都市政府发布的《成都市规范行政处罚自由裁量权实施办法》（市政府令第169号）同时废止。",
    "填报时间": "2014-10-16",
    "责任单位": "市政府办公厅",
    "文　　号": "政府令第185号",
    "签发单位": "",
```

```
    "签发时间": "2014-09-29",
    "生效时间": "2014-09-29"
}
```

更多类似的数据和相应的代码请从 www.broadview.com.cn 网站下载，这里就不一一列出了。以上请求如果成功，则会返回类似于下面的响应（_id 和请求 URL 结尾一致）：

```
HTTP/1.1 201 Created
Location: /chengdugov/fulltext/1
content-type: application/json; charset=UTF-8
content-length: 147

{"_index":"chengdugov","_type":"fulltext","_id":"1","_version":1,"result":"
created","_shards":{"total":2,"successful":1,"failed":0},"created":true}
```

如果想确认以上文档是否存入了 ElasticSearch 中，则可以使用如下命令来验证返回的 JSON 对象：

```
GET http://localhost:9200/chengdugov/fulltext/$id
```

搜索功能

现在，ElasticSearch 已经存储了一些数据，我们可以用 ElasticSearch 的搜索 API 返回符合查询条件的结果。

如果要搜索所有文件，可以使用最简单的搜索命令_search：http://localhost 9200/chengdugov/fulltext/_search，其中，chengdugov 是索引，fulltext 是类型，但是没有指定文档 ID，只是使用了_search 功能，在我们给出的例子中，在返回的 JSON 字符串的命中列表里有 7 个完整的文件。

接下来，我们来搜索哪些文件的题目包含“社会保险”。

```
POST http://localhost:9200/chengdugov/_search
{
    "query": {
        "bool": {
            "must": [{
                "wildcard": {
                    "题目.keyword": "*社会保险*"
```

```
                }
            }],
            "must_not": [],
            "should": []
        }
    },
    "from": 0,
    "size": 10,
    "sort": [],
    "aggs": {

    }
}
```

这个查询包含了复杂的json字符串，如果每次请求都这么麻烦，用户体验就太差了，所以我们利用第三方插件 elasticssearch-head 来帮助查询。下面安装 elasticsearch-head。

(1) 打开命令行窗口，进入目标目录中，输入 git clone git://github.com/mobz/elastic-search-head.git 命令，将 elasticsearch-head 的源码下载到本地。

(2) 执行 cd elasticsearch-head 命令，进到 elasticsearch-head 目录中。

(3) 执行 npm install 命令，安装 packages.json 里的所有包。

(4) 执行 grunt server 命令，启动服务器。

(5) 在浏览器地址栏中输入 http://localhost:9100/，打开 elasticsearch-head 插件的主页面（注意，端口号是 9100，和之前 ElasticSearch 的端口号 9200 不一样，这两个页面独立运行，但是 9100 已经封装好了很多功能，以至于我们不需要手动发送请求）。

在 elasticsearch-head 主页中，第一行填入 http://localhost:9200/，单击 Connect 按钮，连接上 ElasticSearch，同时显示 ElasticSearch 集群的健康状况，如图2.3所示。



图 2.3　集群状态图

如果页面中显示“cluster health: not connected”，请查看浏览器控制台的输出，若看到以下错误：

```
XMLHttpRequest cannot load http://localhost:9200/_cluster/health. No 'Access
-Control-Allow-Origin' header is present on the requested resource. Origin '
http://localhost:9100' is therefore not allowed access.
```

这时需要重新设置 ElasticSearch，修改 elasticsearch-rtf\config 目录中的 ElasticSearch.yml 文件，在文件末尾添加如下两行：

```
1 http.cors.enabled: true
2 http.cors.allow-origin: "*"
```

然后重启 ElasticSearch 服务，应该一切 OK。

第二行显示了 5 个标签。

(1) Overview，显示了健康的 Node 和未使用的 Node，如图2.3所示。

(2) Indices，显示了所有创建的索引，如图2.4所示。



图 2.4　索引列表

(3) Browser（浏览栏状态），显示了每个索引的所有类型和文件，如图2.5所示。

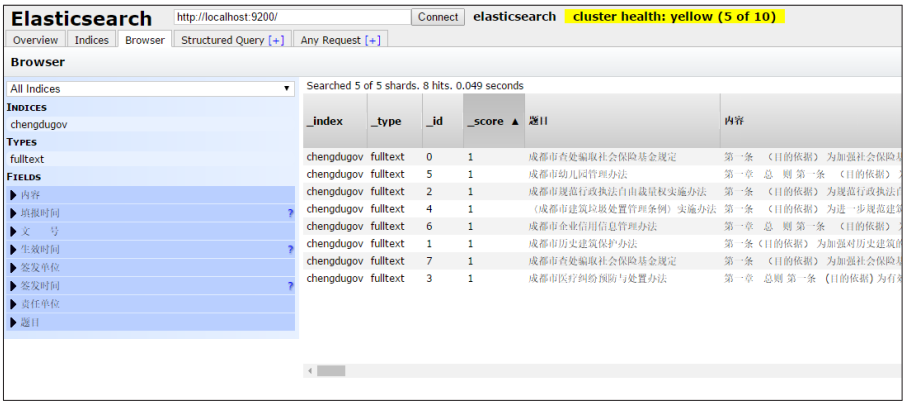


图 2.5　浏览栏状态

(4) Structured Query（结构化查询），提供了所有的搜索功能。比如要搜索题目中包含“社会保险”关键词的文件，使用的命令如图2.6所示。



图 2.6　结构化查询

该查询返回两条结果。同时，通过查看浏览器调试工具，可以看到如图2.7所示的请求列表。

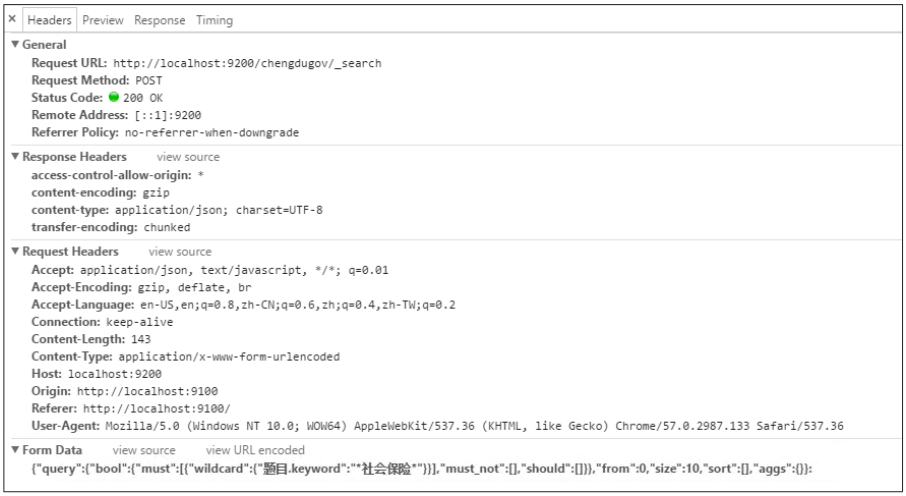


图 2.7　浏览器接收到的查询信息

(5) AnyRequest（任意请求），这个标签为开发者提供了更多的选择来构造不同的请求（网址、http 方法、body）。还是以上一个搜索为例子，把 body 粘贴到框里，会得到相同的结果，如图2.8所示。

段落（文件）摘要功能的实现

现在，我们在 ElasticSearch 的基础上实现段落摘要的功能。段落内容往往很长，所以有需求——生成简短的摘要，一目了然。段落摘要功能和 ElasticSearch 没有直接关系，只是借助 ElasticSearch 结构化存储而已。这里有一个基本概念需要介绍，称为 TD-IDF。

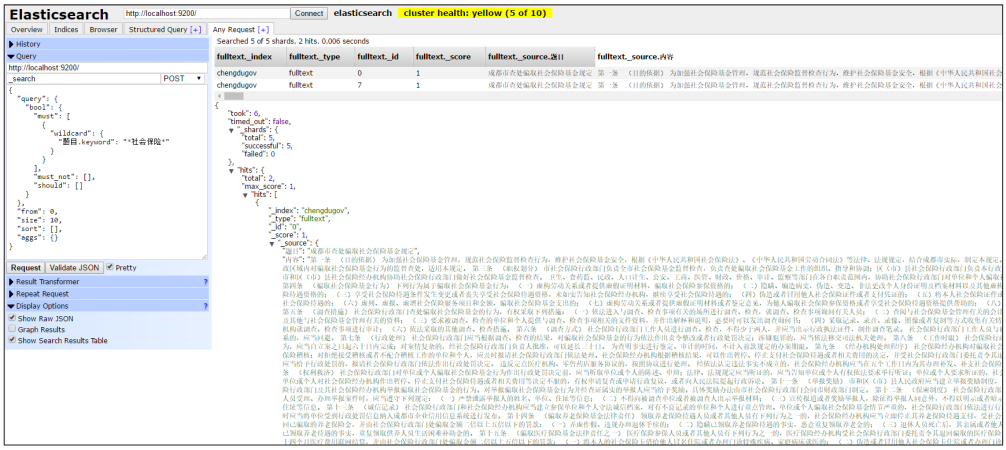


图 2.8　任意请求

TF-IDF（Term Frequency-Inverse Document Frequency）是一种用于进行信息检索与数据挖掘的常用加权技术，用以评估一个词对于一个段落集或一个语料库中的其中一个段落的重要程度。词的重要性随着它在段落中出现的次数成正比增加，但同时会随着它在语料库中出现的频率成反比下降。TF-IDF 加权的各种形式常被搜索引擎应用，作为段落与用户查询之间相关程度的度量或评级。除 TF-IDF 以外，因特网上的搜索引擎还会使用基于链接分析的评级方法，以确定段落在搜寻结果中出现的顺序。

有很多不同的数学公式可以用来计算 TF-IDF。下面要讲述的例子用最常见的公式来计算。

首先，词频（TF）是一个词出现的次数除以该段落的总词数。假如一个段落的总词数是 100 个，而“政策”一词出现了 6 次，那么“政策”这个词在该段落中的词频就是 6/100=0.06。

其次，计算段落频率（IDF）的方法是一个段落集里包含的段落总数除以出现过“政策”一词的段落的数量，通常取对数以消除长尾效应。所以，如果“政策”一词在 20 000 个段落里出现过，而段落总数是 20 000 000 个，其逆向段落频率就是 log(20 000 000/20 000) = 3。最后 TF-IDF 的分数为 0.06 × 3 = 0.18。

Word2Vec 是 Google 推出的用来进行词的向量表达的开源工具包，这个名字也是该工具包所代表的算法的称号。在自然语言处理中，我们把一个句子当成词的集合，那么一篇文章中所出现的所有词的集合就称之为“词典”（vocabulary）。Word2Vec 的主要思想是把词表达为低维度向量的形式，含义相近的词在这个低维度向量空间中的位置相近，而不相关的词则距离较远。

为了执行下面的摘要程序，需要使用 scipy、gensim 库，如果在 Python 环境中没有安装这些库的话，则可以使用 pip install scipy,gensim 命令来安装。如果使用的是 Anaconda Python，那么 scipy 已经安装了，只需安装 gensim 库即可，在命令行输入 conda install gensim 进行安装。

首先加载需要用到的库。

```
import os, json, gensim,  requests, math
import numpy as np
```

然后加载已训练好的中文 Word2Vec 词向量表达，即词嵌入。

```
model = gensim.models.Word2Vec.load("wiki.en.text.jian.model")
```

接下来定义余弦相似性函数，计算向量 v1 和 v2 的 Cosine 乘积。

```
def cosine_sim(v1,v2):
    sumxx, sumxy, sumyy = 0, 0, 0
    for i in range(len(v1)):
        x = v1[i]; y = v2[i]
        sumxx += x*x
        sumyy += y*y
        sumxy += x*y
    return sumxy/math.sqrt(sumxx*sumyy)
```

下面把传入的所有段落列表使用 ElasticSearch RTF 自带的分词器进行分词，用来构建词典。

```
def get_vocabulary(aList):
    url = 'http://localhost:9200/chengdugov/_analyze?analyzer=ik_max_word'
    headers = {}
    arrayofTokenArray = []
    mergedTokenArray = []
    for i in range(len(aList)):
        res = requests.post(url, data = aList[i], headers = headers)     # 返
        回API结果
        tokens = [json.loads(res.text)['tokens'][i]['token'] for i in range(
        len(json.loads(res.text)['tokens']))]
        arrayofTokenArray.append(tokens)    # 每个段落以分词的形式表示
        mergedTokenArray.extend(tokens)     # 所有段落里的分词集合
```

```
    vocablist = list(set(mergedTokenArray))# 所有段落里的分词去重以后的集合
    vocabulary_dict = {}
    for i in range(len(vocablist)):
        vocabulary_dict[vocablist[i]] = i    # 构建词典，key是分词，value是
        该分词在所有段落中出现的次数
    return vocabulary_dict, arrayofTokenArray
```

确定组成段落的词的 tfidf 权重矩阵。

```
def get_tfidf(voc_d, t):
  m = np.zeros((len(t), len(voc_d)))    # 构造段落分词矩阵，行为段落，列为分
  词，每个格子代表分词在该段落中出现的次数。
  # 每个段落对应一行，由一个维度为词典长度的权重向量表达。向量第一个起始下标
  为0
  for i in range(len(t)):
    listofIndexofwordsAppearedinEach = [voc_d[item] for item in t[i]]
    for item in listofIndexofwordsAppearedinEach:
      m[i][item] += 1
  transformer = TfidfTransformer()
  result = transformer.fit_transform(m).toarray()    # 计算每个段落的每个分
  词的tfidf
  return result
```

把段落中的词用 Word2Vec 词向量表达，并结合 TF-IDF 权重，算出段落的向量表达。

```
def get_all_embedding(t, result, voc_d, model):
  emb = []
  for i in range(len(t)):
    vec = np.zeros(400)    # 每个段落用维度为400的向量表达，是我们最终想要的
    段落嵌入。向量第一个起始下标为0
    for item in t[i]:
      try:
        newvec = model[item]*result[i][voc_d[item]]
        vec += newvec    # 每个段落的向量表达，即段落嵌入是其分词对应的
        Word2Vec词向量依该分词的TF-IDF权重的加权平均
      except KeyError:
        pass
    emb.append(vec)
```

```
    return emb    # 返回所有段落的向量表达

def getTitlesandContent():
    titles = []
    contents = []
    for i in range(7):    # 这里为了阐述，数据集中有7个标题和段落
        url = 'http://localhost:9200/chengdugov/fulltext/' + str(i)
        res = requests.get(url)
        title = json.loads(res.text)['_source']['题目']
        content = json.loads(res.text)['_source']['内容']
        titles.append(title)    # 获取7个标题
        contents.append(content)    # 获取7个段落的具体文字
    return titles, contents
```

下面定义基于 Cosine 的相似度计算函数。

```
def get_similar_1(i, glb_emb, emb):
  cor = [cosine_sim(glb_emb[i], emb[j]) for j in range(len(emb))]
  # 计算段落中每句话的向量表达和段落向量表达的相似度，以实现摘要功能，即选取和段落意思最接近的句子。
  return cor
```

我们看到，段落摘要的思想是，对于任何一个段落，都可以得出该段落的向量表达，也可以得出其中每个句子的向量表达。段落摘要相当于与该段落相关性最高的段落内句子的集合。

```
def get_abstract(i, topN, contentsX, glb_emb):    # 给定段落集合，选取第i个段落中与段落意思最接近的N个句子的集合
    print(contentsX[i])
    paragraphsArray = contentsX[i].decode("utf-8").replace(';', '。').split('。')[:-1]
    paragraphsArray = [t.encode("utf-8") for t in sentencesArray]  # 段落集合
    voc_d, t = get_vocabulary(paragraphsArray)    # 该段落中所有分词的不重复集合和段落中每个句子的分词集合
    result = get_tfidf(voc_d,t)    # 该段落中每个句子的分词权重矩阵
    emb = get_all_embedding(t, result, voc_d, model)    # 该段落中每个句子的向量表达，即句子中分词的向量依TF-IDF权重的加权平均
```

```
    cor = get_similar_1(i, glb_emb, emb)    # 计算该段落中每个句子的向量表达和该段落向量表达的相似性
    s = sorted(range(len(cor)), key = lambda i: cor[i])[(-topN):]    # 排序找出最接近该段落的N个句子。
    return '。'.join([sentencesArray[item].decode("utf") for item in s])
    # 返回N个句子的集合
```

下面是主要的调用程序入口。

```
titles, contents = getTitlesandContent()    # 获取数据集中的所有段落和标题。这里为便于阐述，数据集中有7个段落及相应的标题
contentsX = [t.encode("utf-8") for t in contents]
glb_voc_d, glb_t = get_vocabulary(contentsX)    # 获取所有分词的不重复集合和每个段落中的分词集合
glb_result = get_tfidf(glb_voc_d,glb_t)    # 获取段落分词权重矩阵
glb_emb = get_all_embedding(glb_t, glb_result, glb_voc_d, model)    # 获取每个段落的嵌入表达
abs = get_abstract(2, 7, contentsX)    # 对第2个段落取出最接近这个段落主旨的7个句子作为摘要
print(abs)
```

3 深度学习简介

3.1 概述

深度学习是现在非常热门的机器学习和人工智能工具，具备了很多以前觉得难以实现的学习功能。深度学习本身是传统神经网络算法的延伸，而传统神经网络模型的历史甚至可以追溯到 20 世纪 50 年代，现在公认其鼻祖是 Rosenblatt 在 1957 年提出的感知器（Perceptron）算法。到目前为止，神经网络模型的发展大概经历了四个不同的起伏阶段。

第一个阶段是 20 世纪 50 到 60 年代，这个时候的神经网络模型还属于基本的感知器，非常简单。第二个阶段是 20 世纪 70 到 80 年代，在这个阶段，多层感知器（Multilayer Perceptron）被发现，其逼近高度非线性函数的能力使得科学界对它的兴趣大增，甚至有神经网络能解决一切问题的论调，这可以算作神经网络模型的一个高潮。第三个阶段是 20 世纪 90 年代到 21 世纪早些时候，这个阶段基本上是传统神经网络模型比较沉寂的时期，但却是核方法（Kernel）大行其道的时候。主要原因是计算能力跟不上，还有就是大规模的数据在这个前互联网时代并不是随处可见的，因此神经网络模型无论是从计算还是性能角度都不能跟传统的机器学习方法相提并论。第四个阶段是大约 2006 年以后到现在，这个时期有几个重要的技术进步促进了以深度学习为代表的神经网络模型的大规模应用。首先是廉价的并行计算的出现，比如 GPGPU 概念的出现；其次是深度网络结构的持续研究，使得模型训练效率大大增加；最后是互联网的出现，为大规模数据的生成和获取提供了极大的便利。这些因素能够充分发挥深度神经网络无限逼近高度非线性函数的普适性，同时深度学习架构的灵活性使得这类模型稍加修改就能

用来解决不同的问题，从而使其应用范围大大增加。总之，计算的便利性和预测质量的提高是神经网络模型再次得到青睐的主要原因。

一般来说，深度学习适合解决数据量大、数据比较规范，但是决策函数高度非线性的问题。现在常见的深度学习应用非常成功的领域有图像识别、语音识别、文字生成、自然语言分析等。这几类应用的共同特点是数据量极大，同时其数据都是工程应用的输出，具备较好的多样性和规范性；另外，其决策函数都是高度非线性、高度复杂的。比如在图像识别中，输入的数据是每个像素点的位置和其对应的颜色，这些数值都出现在一个有限的区域中，并且在不同的值域中都有大量的数据覆盖供模型学习。同时，要识别在图像中不同位置出现的同一类事物，需要非常复杂的决策函数，这在传统的机器学习方法里是不容易实现的。

比如常见的手写数字识别，如果单纯采用欧式空间距离输入到传统的机器学习算法里来判断是否跟某一个数字类似，则会受到字符的旋转、位置不齐等因素的影响，造成分类器质量不高。传统的机器学习通过引入非线性变换，比如旋转和位移，能够部分解决这些问题。但是深度学习通过采用高度非线性的手段，对图像进行切割比较，能够有效地克服上述困难，获得比普通方法更好的结果，同时这种方法具备一定的普遍性，不仅能处理数字识别，稍加修改还能处理其他事物的识别，而不像传统手段对于每一类具体问题都可能需要设计全新的方法，因此在应用效率上也高很多。

下面我们就简要介绍深度学习的基本知识。

3.2 深度学习的统计学入门

一般书籍对深度学习的入门介绍总是直接从有向图或者解析神经网络的目标函数开始，有大量的数学推导，这其实不利于应用型读者理解深度学习的本质。在这一节中，我们通过一个传统的统计学建模例子来帮助读者快速理解深度学习多层网络的概念，并与现有的知识结构有机联系起来。

这个例子使用了著名的鸢尾花卉数据集，图3.1展示了鸢尾属下的三个亚属，即 Iris Setosa（山鸢尾，简称 S）、Iris Versicolour（杂色鸢尾，简称 C）和 Iris Virginica（维吉尼亚鸢尾，简称 V）之间花萼和花瓣的长度关系。

我们可以看到在不同的亚属之间，花萼和花瓣的长度关系并不一样。山鸢尾的花萼和花瓣之间只有很弱的相关性，而其他两种亚属则具有很强的相关性，不过杂色鸢尾的花萼和花瓣长度均稍小于维吉尼亚鸢尾。

在这种情况下，如果直接使用花萼长度对花瓣长度建模，效果会比较差，但是给定的鸢尾花亚属，花萼和花瓣的长度关系还是比较强的。因此在传统的统计建模中，分析

师会采用将鸢尾花亚属作为一个分类变量，并与花萼长度做一个交叉项来建模。如果采用所谓的 GLM（Generalized Linear Model，广义线性模型）编码法，不同亚属会分别得到各自不同的斜率，或者说权重；同时截距项也可以分别得到不同亚属对应的估计值。GLM 编码法在机器学习领域被称为 One Hot 编码法。

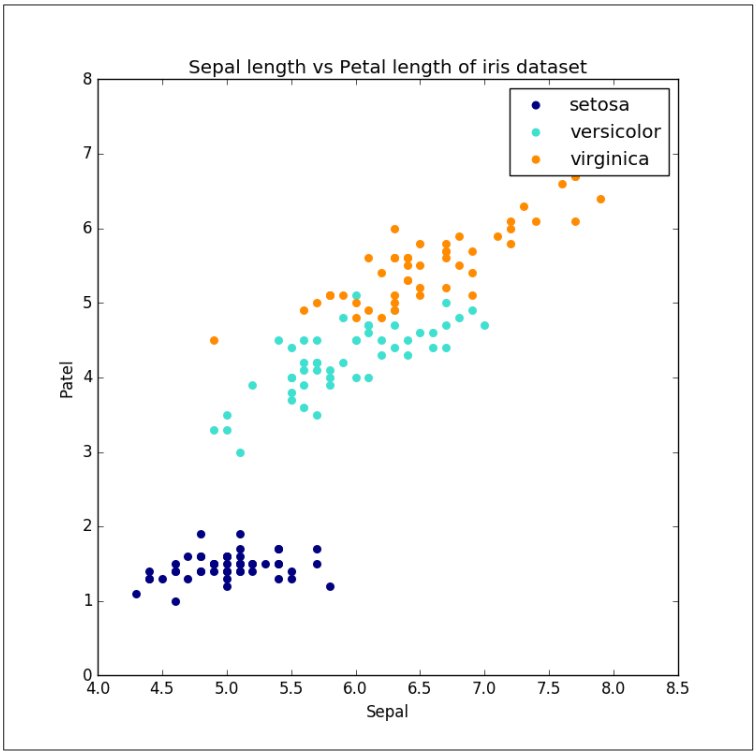


图 3.1　鸢尾属下的三个亚属的花萼和花瓣的长度关系

如果仔细考察计算公式，这相当于将数据按照亚属拆分成三块，然后分别对花萼和花瓣的长度关系进行建模，最后再给不同亚属分配不同的亚属权重来调整单独建模后得到的斜率。

$$E(y) = \alpha_0 * c + \alpha_1 * l + \alpha_2 * c * l$$
$$= \alpha_0 * c + (\alpha_1 + \alpha_2 * c) * l$$

其中，y 是花瓣长度，c 对应于鸢尾花亚属，l 是花萼长度，$c * l$ 是交叉项。

按照 One Hot 编码法对亚属的分类设定数值，那么鸢尾花亚属分类变量对应的数值矩阵可以写为表3.1所示的形式。

将这个矩阵带入上面的公式中，再合并同属类的项目，则公式变为：

$$E(y) = (\alpha_s + \beta_s * l) + (\alpha_c + \beta_c * l) + (\alpha_v + \beta_v * l)$$

表 3.1　鸢尾花亚属 One Hot 编码结果

鸢尾花亚属	山鸢尾（S）	杂色鸢尾（C）	维吉尼亚鸢尾（V）
山鸢尾（S）	1	0	0
山鸢尾（S）	1	0	0
杂色鸢尾（C）	0	1	0
杂色鸢尾（C）	0	1	0
维吉尼亚鸢尾（V）	0	0	1

换句话说，每个亚属对应的数据都可以用来构造一个决策函数，比如 $(\alpha_v + \beta_v * l)$，但是因为是分别决策的，不足以优化全局的损失函数，因此需要将这些独立的决策函数再纳入一个更高层级的决策函数中，来最后优化总的损失函数。在上面的线性公式中，这个全局优化的权重在估计的过程中使用的都是 Hessian 矩阵的对角线项目，但是因为模型提前假设为没有异方差性，因此每一个亚属构造的决策函数具有相同的权重。这三个亚属的决策函数，可以分别看成一个隐含层的节点，即神经元。我们可以将每个数据点带入一个包含每个亚属的计算表格中进行计算，如表3.2所示。

表 3.2　隐含层计算表格

数据点	山鸢尾（S） 决策函数	杂色鸢尾（C） 决策函数	维吉尼亚鸢尾（V） 决策函数	输出
$x : c = \mathrm{S}, l = 5.1$	$h_s(x; \alpha_s, \beta_s)$	0	0	$\hat{h}_s(x)$
$x : c = \mathrm{S}, l = 4.9$	$h_s(x; \alpha_s, \beta_s)$	0	0	$\hat{h}_s(x)$
$x : c = \mathrm{C}, l = 7.2$	0	$h_c(x; \alpha_c, \beta_c)$	0	$\hat{h}_c(x)$
$x : c = \mathrm{C}, l = 6.2$	0	$h_c(x; \alpha_c, \beta_c)$	0	$\hat{h}_c(x)$
$x : c = \mathrm{V}, l = 5.9$	0	0	$h_v(x; \alpha_v, \beta_v)$	$\hat{h}_v(x)$

根据这个计算表格，这个过程可以很自然地用网络图来表示，如图3.2所示。

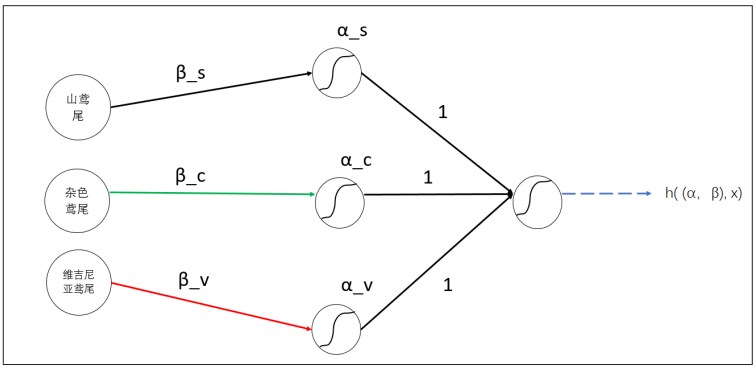


图 3.2　鸢尾花统计模型的网络图

但是因为使用了线性激活函数（Identity Activation Function），而线性函数的函数仍然是一个线性函数，因此图3.2中的隐藏层通常被简化为图3.3中所显示的一个节点。

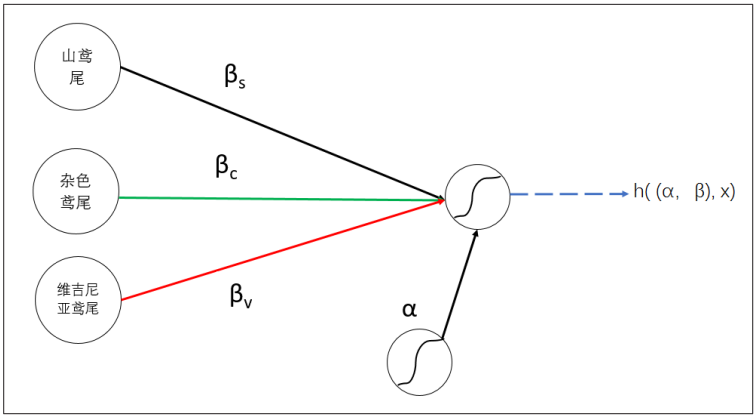


图 3.3　简化后的鸢尾花统计模型的网络图

当然，这只是最简单的情形，不过有效地说明了传统统计方法和神经网络之间的密切联系。上述例子使用了线性函数，只有为数不多的几个参数，但是在实际中一般都拓展为非线性函数，并且参数的空间较大，然而其整体结构是一致的。一个深度学习的神经网络可以被表述为如图3.4所示的形式。

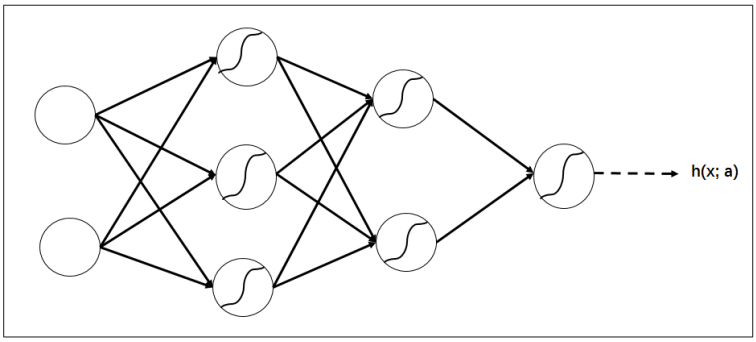


图 3.4　神经网络的基本构造形态

3.3　一些基本概念的解释

要理解深度学习，特别是正确地运用 Keras 来构造自己的神经网络模型，并应用到实际业务中，有必要对深度学习中一些常见的概念进行略微深入的了解。构造 Keras 代码时基本是围绕这些概念来进行的，因此了解这些概念能帮助理解下一章将要介绍的

Keras 命令。图3.5集中展示了一部分基本概念的相对关系，概念本身用中文和虚线箭头标注。

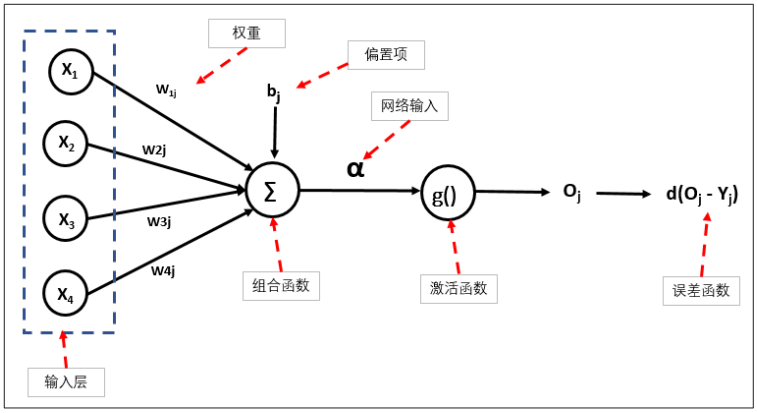


图 3.5　神经网络的基本构件

3.3.1　深度学习中的函数类型

大多数神经网络中都包含四类函数：组合函数（Combination Function）、激活函数（Activation Function）、误差函数（Error Function）和目标函数（Object Function）。下面就简单介绍每类函数的作用和目的。

组合函数

在神经网络中，在输入层之后的网络里，每个神经元的功能都是将上一层产生的向量通过自身的函数生成一个标量值，这个标量值就称为下一层神经元的网络输入变量。这种在网络中间将向量映射为标量的函数就被称为组合函数。常见的组合函数包括线性组合函数和基于欧式空间距离的函数，比如在 RBF 网络中常用的函数。

激活函数

大多数神经元都将一维向量的网络输入变量通过一个函数映射为另外一个一维向量的数值，这个函数称为激活函数，其产生的值就被称为激活状态。除输出层以外，激活状态的值通过神经网络的链接输入到下一层的一个或者多个神经元里面。这些激活函数通常都是将一个实数域上的值映射到一个有限域中，因此也被称为塌缩函数。比如常见的 tanh 或者 logistic 函数，都是将无限的实数域上的数值压缩到 $(-1, 1)$ 或者 $(0, 1)$

之间的有限域中。如果这个激活函数不做任何变换，则被称为 Identity 或者线性激活函数。

激活函数的主要作用是为隐含层引入非线性。一个只有线性关系隐含层的多层神经网络不会比一般的只包含输入层和输出层的两层神经网络更加强大，因为线性函数的函数仍然是一个线性函数。但是加入非线性以后，多层神经网络的预测能力就得到了显著提高。对于后向传播算法，激活函数必须可微，而且如果这个函数是在有限域中的话，则效果更好，因此像 logistic、tanh、高斯函数等都是比较常见的选择，这类函数也被统称为 sigmoid 函数。类似于 tanh 或者 arctan 这样的包含正和负的值域的函数通常收敛速度较快，因为数值条件数（Conditioning Number）更好。

对于隐藏层而言，流行的激活函数经历了从 sigmoid 激活函数到 threshold 激活函数的转变，这反映了深度学习技术和理论的发展。

早期的理论认为 sigmoid 激活函数通常比 threshold 激活函数（比如 ReLU 等激活函数）好。理由是因为采用 threshold 激活函数后误差函数是逐级常数（Stepwise Constant），因此一阶导数要么不存在要么为 0，从而不能使用高效的后向传播算法来计算一阶导数（Gradient）。即使使用那些不采用梯度的算法，比如 Simulated Annealing 或者基因算法，采用 sigmoid 激活函数在传统上仍然被认为是一个较好的选择，因为这种函数是连续可微的，参数的一点变化就会带来输出的变化，这有助于判断参数的变动是否有利于最终目标函数的优化。如果采用 threshold 激活函数，参数的微小变化并不能在输出中产生变动，因此算法收敛会慢很多。

但是近期深度学习模型的发展改变了这些观点。sigmoid 函数存在梯度消亡（Gradient Vanishing）的问题。这个问题是由 Sepp Hochreiter 在其 1991 年发表的硕士论文中正式提出的。梯度消亡指的是梯度（误差的信号）随着隐藏层数的增加成指数减小。这是因为在后向传播算法中，对梯度的计算使用链式法则，因此在第 n 层时需要将前面各层的梯度都相乘，但是由于 sigmoid 函数的值域在 $(-1,1)$ 或者 $(0,1)$ 之间，因此多个很小的数相乘以后第 n 层的梯度就会接近于 0，造成模型训练的困难。而 threshold 激活函数因为值域不在 $(-1,1)$ 之间，比如 ReLU 的取值范围是 $[0,+\text{inf})$，因此没有这个问题。

另外一些 threshold 函数，比如 Hard Max 激活函数：$\max(0,x)$，可以在隐藏层中引入稀疏性（Sparsity），也有助于模型的训练。

对于输出层，读者应该尽量选择适合因变量分布的激活函数。

- 对于只有 0,1 取值的双值因变量，logistic 函数是一个较好的选择。
- 对于有多个取值的离散因变量，比如 0 到 9 数字的识别，softmax 激活函数是 logistic 激活函数的自然衍生。

- 对于有有限值域的连续因变量，logistic 或者 tanh 激活函数都可以用，但是需要将因变量的值域伸缩到 logistic 或者 tanh 对应的值域中。
- 如果因变量取值为正，但是没有上限，那么指数函数是一个较好的选择。
- 如果因变量没有有限值域，或者虽然是有限值域但是边界未知，那么最好采用线性函数作为激活函数。

我们看到，在输出层的这些激活函数，其选择跟对应的统计学模型的应用有类似的地方。读者可以将这个关系理解为统计里的广义线性模型中的联结函数（Link Function）的功能。

误差函数

监督学习的神经网络都需要一个函数来测度模型输出值 p 和真实的因变量值 y 之间的差异，甚至有些无监督学习的神经网络也需要类似的函数。模型输出值 p 和真实值 y 之间的差异一般被称为残差或者误差，但是这个值并不能直接用来衡量模型的质量。当一个模型完美的时候（虽然这不可能出现），其误差为 0，而当一个模型不够完美的时候，其误差不论为负值还是正值，都偏离 0；因此衡量模型质量的是误差偏离 0 的相对值，即误差函数的值越接近于 0，模型的性能越好，反之则模型的性能越差。误差函数也被称为损失函数。常用的误差函数如下。

- 均方差（MSE）：$\frac{1}{N}\sum_i (y_i - p_i)^2$，这种损失函数通常用在实数值域连续变量的回归问题上，并且对于残差较大的情况给予更多的权重。
- 平均绝对差（MAE）：$\frac{1}{N}\sum_i |y_i - p_i|$，这种损失函数也通常用在上面提到的那类回归问题上，在时间序列预测问题中也常用。在这个误差函数中，每个误差点对总体误差的贡献与其误差绝对值成线性比例关系，而上面介绍的 MSE 没有这个特性。
- 交叉熵损失（Cross-Entropy）：这种损失函数也叫作对数损失函数，是针对分类模型的性能比较设计的，按照分类模型是二分类还是多分类的区别，可以分成二分类交叉熵和多分类交叉熵两种。交叉熵的数学表达式很简单，可以写作：

$$J(\theta) = -\left[\sum_{i=1}^{N}\sum_{k=1}^{K} 1(y^{(i)} = k)\log P(y^{(i)} = k|x^{(i)};\theta)\right]$$

因此交叉熵可以被解释为映射到最可能的类别的概率的对数。因此，当预测值的分布和实际因变量的分布尽可能一致时，交叉熵最小。

目标函数

目标函数是需要在训练阶段直接最小化的那个函数。神经网络的训练表现为在最小化训练集上估计值和真实值之间的误差。其结果很可能出现过度拟合的现象，即模型在训练集上表现较好，但是在测试集上和其他真实应用中表现不好，即所谓的模型普适化不好。一般这时会采用正则化来规范模型，减少过度拟合情况的出现。这个时候目标函数为误差函数和正则函数的和。比如使用了权重递减（Weight Decay）的方法，正则函数为权重的平方和，这和一般的岭回归（Ridge Regression）使用的技巧一样。如果运用贝叶斯的思路，也可以将权重的先验分布的对数作为正则项。当然，如果不采用正则项，那么目标函数就和总的或者平均误差函数一样了。

3.3.2 深度学习中的其他常见概念

批量

批量，即 Batch，是深度学习中的一个重要概念。批量通常指两个不同的概念——如果对应的是模型训练方法，那么批量指的是将所有数据处理完以后一次性更新权重或者参数的估计；如果对应的是模型训练中的数据，那么批量通常指的是一次输入供模型计算用的数据量。这两个概念有着紧密的关系。

基于批量概念的模型训练通常按照如下步骤进行。

(1) 初始化参数。

(2) 重复以下步骤。

- 处理所有数据。
- 更新参数。

和批量算法相对应的是递增算法，其步骤如下：

(1) 初始化参数。

(2) 重复以下步骤。

- 处理一个或者一组数据点。
- 更新参数。

我们看到，这里的主要区别是批量算法一次处理所有的数据；而在递增算法中，每处理一个或者数个观测值就要更新一次参数。这里“处理”和“更新”二词根据算法的不同有不同的含义。在后向传播算法中，“处理”对应的具体操作就是计算损失函数的梯度变化曲线。如果是批量算法，则计算平均或者总的损失函数的梯度变化曲线；而如

果是递增算法，则计算损失函数仅在对应于该观测值或者数个观测值时的梯度变化曲线。“更新”则是从已有的参数值中减去梯度变化率和学习速率的乘积。

在线学习和离线学习

在深度学习中，另外两个常见的概念是在线学习（Online Learning）和离线学习（Offline Learning）。在离线学习中，所有的数据都可以被反复获取，比如上面的批量学习就是离线学习的一种。而在在线学习中，每个观测值在处理以后会被遗弃，同时参数得到更新。在线学习永远是递增算法的一种，但是递增算法却既可以离线学习也可以在线学习。

离线学习有如下几个优点。

- 对于任何固定个数的参数，目标函数都可以直接被计算出来，因此很容易验证模型训练是否在朝着所需要的方向发展。
- 计算精度可以达到任意合理的程度。
- 可以使用各种不同的算法来避免出现局部最优的情况。
- 可以采用训练、验证、测试三分法对模型的普适度进行验证。
- 可以计算预测值及其置信区间。

在线学习无法实现上述功能，因为数据并没有被存储，不能反复获取，因此对于任何固定的参数集，无法在训练集上计算损失函数，也无法在验证集上计算误差。这就造成在线算法一般来说比离线算法更加复杂和不稳定。但是离线递增算法并没有在线算法的问题，因此有必要理解在线学习和递增算法的区别。

偏移/阈值

在深度学习中，采用 sigmoid 激活函数的隐藏层或者输出层的神经元通常在计算网络输入时加入一个偏移值，称为 Bias。对于线性输出神经元，偏移项就是回归中的截距项。

跟截距项的作用类似，偏移项可以被视为一个由特殊神经元引出的链接权重，这是因为偏移项通常链接到一个取固定单位值的偏移神经元。比如在一个多层感知器（MLP）神经网络中，某一个神经元的输入变量为 N 维，那么这个神经元在这个高维空间中根据参数画一个超平面，一边是正值，一边为负值。所使用的参数决定了这个超平面在输入空间中的相对位置。如果没有偏移项，这个超平面的位置就被限制住了，必须通过原点；如果多个神经元都需要其各自的超平面，那么就严重限制了模型的灵活性。这就好比一个没有截距项的回归模型，其斜率的估计值在大多数情况下会大大偏移最

优估计值，因为生成的拟合曲线必须通过原点。因此，如果缺少偏移项，多层感知器的普适拟合能力就几乎不存在了。

通常来说，每个隐藏层和输出层的神经元都有自己的偏移项。但是如果输入数据已经被等比例转换到一个有限值域中，比如 [0, 1] 区间，那么第一个隐藏层的神经元设置了偏移项以后，后面任何层跟这些具备偏移项的神经元有链接的其他神经元就不需要再额外设置偏移项了。

标准化数据

在机器学习和深度学习中，常常会出现对数据标准化这个动作。那么什么是标准化数据呢？其实这里是用“标准化”这个词代替了几个类似的但又不同的动作。下面详细讲解三个常见的“标准化”数据处理动作。

(1) 重放缩（Rescaling）：通常指将一个向量加上或者减去一个常量，再乘以或者除以一个常量。比如将华氏温度转换为摄氏温度就是一个重放缩的过程。

(2) 规范化（Normalization）：通常指将一个向量除以其范数，比如采用欧式空间距离，则用向量的方差作为范数来规范化向量。在深度学习中，规范化通常采用极差为范数，即将向量减去最小值，并除以其极差，从而使数值范围在 0 到 1 之间。

(3) 标准化（Standardization）：通常指将一个向量移除其位置和规模的度量。比如一个服从正态分布的向量，可以减去其均值，并除以其方差来标准化数据，从而获得一个服从标准正态分布的向量。

那么在深度学习中是否应该进行以上任何一种数据处理呢？答案是依照情况而定。一般来讲，如果激活函数的值域在 0 到 1 之间，那么规范化数据到 [0, 1] 的值域区间是比较好的。另外一个考虑是规范化数据能使计算过程更加稳定，特别是在数据值域范围区别较大的时候，规范化数据总是相对稳健的一个选择。而且很多算法的初始值设定也是针对使规范化以后的数据更有效来设计的。

3.4 梯度递减算法

在对决策函数进行优化的时候，通常是针对一个误差的度量，比如误差的平方，以求得一系列参数，从而最小化这个误差度量的值来进行的，而目前一般采用的计算方法是梯度递减法（Gradient Descent Method）。这是一个非常形象的名字，好比一个游客要从某个不知名的高山上尽快、安全地下到谷底，这时候需要借助指南针来引导方向。对于这个游客，他需要在南北和东西两个轴向上进行选择，以保证下山的路在当前环

境下既是最快的又是安全的。读者可以把南北和东西两个轴向想象成目标函数里面的两个维度或者自变量。那么这个游客怎么获取这个最优的路径呢?

在山顶的时候，游客因为不能完全看到通往谷底的情况，所以很可能随机选择一条线路。这个选择很多时候很关键。一般山顶是一块平地，有多个可以选择下山的可能点。如果真正下山的路线是在某一个地方，而游客选择了另外一个地方，则很可能最后到不了真正的谷底，可能到达半山腰或者山脚下的某一个地方，但是离真正的谷底差距可能不小。这就是优化问题中的由于初始化参数不佳导致只能获得局部最优解的情况。图3.6形象地展示了这样的情况。

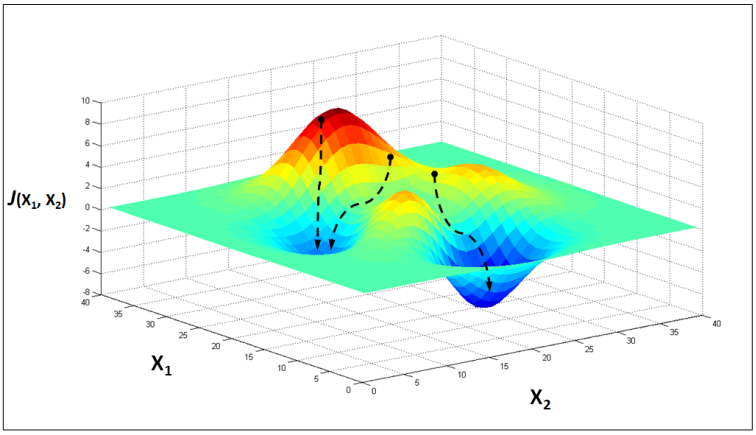


图 3.6　在优化算法中初始值的影响

梯度递减法是一种短视的方法，好比游客在下山的时候遇到非常浓的大雾，只能看见脚下一小块地方，游客就把一个倾角计放在地上，看哪个方向最陡，然后朝着最陡的方向往下滑一段距离，下滑的距离通常根据游客对当前地形的审时度势来决定，停下来，再审视周边哪个方向现在是最陡的，继续重复上面的倾角计算并往下滑的动作。这跟优化中常用的最陡下降法很类似，可以看作最陡下降法的一个特例。在最陡下降法中，参数的更新使用如下公式：

$$w_k^{(t+1)} = w_k^{(t)} - \varepsilon \nabla_{w_k} f(w)$$

其中，$w_k^{(t)}$ 是第 k 个参数在 t 次迭代时的值，$\nabla_{w_k} f(w)$ 是误差函数对应于该参数的一阶偏微分，而 ε 则是当前步进的大小，一般通过线性搜索获得一个当前最优值。

但是在用梯度递减法求解神经网络模型时，通常使用的是随机梯度递减法（Stochastic Gradient Descent），其公式如下：

$$w_k^{(t+1)} = w_k^{(t)} - \Delta w_k^{(t+1)}$$
$$\Delta w_k^{(t+1)} = -\alpha \Delta w_k^{(t)} + \varepsilon \nabla_{w_k} f(w)$$

这个算法有如下几点变化。

- 首先，在计算时不是通览所有的数据后再执行优化计算，而是对于每个观测值或者每组观测值执行梯度递减的优化计算。原来的那种算法因此被叫作批量（Batch）或者离线（Offline）算法，而现在这种算法则被称为递增（Incremental）或者在线（Online）算法，因为参数估计值随着观测组的更新而更新。
- 其次，这个步进值通常从一开始就固定为一个较小的值。
- 最后，通过上述公式可以看出，参数更新部分不仅取决于一阶偏微分的大小，还包含了一个动量项 $\alpha \Delta w_k^{(t)}$。这个动量项的效果是将过去的累计更新项的一部分加入到当前参数的更新项中，即把过去每一步的更新做一个指数递减的加权求和，可以看作对过往的更新值的记忆，越远的记忆影响越小，越近的记忆影响越大。这有助于算法的稳定性。如果步进值极小，而动量项里的控制变量 α 接近于数值 1，那么在线算法就近似于离线算法。

图3.7对梯度递减算法进行了形象的展示。

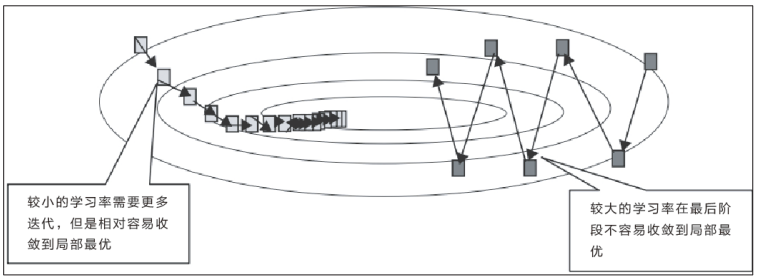


图 3.7　梯度递减算法演示

虽然现在最常见的算法是基于一阶偏微分的梯度递减法，但是跟其他几种以前常用的基于二阶偏微分的优化算法进行比较还是比较有趣的，有助于读者更好地理解这些算法。

基于二阶偏微分的算法通常统称为牛顿法，因为使用了比一阶偏微分更多的信息，可以看作游客在下山的过程中雾小了点，能直接看到周边的地势。假定整座山是一个平滑的凸形状，游客就可以一路下滑到谷底，不用中途停下来。当然，这个谷底也不能保证是最低的，有可能也是某个半山腰的洼地，因为还是有雾，游客无法彻底看清整个地势。

对一般的牛顿法的一种改进叫作增稳牛顿法（Stabilized Newton Method）。这种方法相当于游客带了一个高度计，因此他在滑下去以后可以查看结果，如果发现地势反而增高了，那么游客退回到原来的地方，重新跳一小步，从而保证每次下滑都能到达更低的点。对这种方法的进一步改进叫作岭增稳牛顿法（Ridge Stabilized Newton Method）。这种方法在上一种方法的基础上，游客不仅可以退回到原来的地方，而且重新下滑时还可以选择跳的方向，以保证有更多的机会使得下滑都离谷底更进一步。

对于深度学习模型中的函数，我们看到在每一层的节点都是一个激活函数套着一个组合函数的形式（参见图3.5），即常见的复合函数形式，那么在参数更新部分就需要用到微积分里面的链式法则（Chain Rule）来计算复合函数的导数。假设有函数 $y=J(z)$，$z=g(h)$，$h=h(w)$，那么应用链式法则 $\frac{\partial f}{\partial w}=\frac{\partial f}{\partial h}\frac{\partial h}{\partial w}$，可以得到：

$$\frac{\partial J(w)}{\partial w}=\frac{\partial J}{\partial z}\frac{\partial z}{\partial h}\frac{\partial h}{\partial w}$$

如果假设损失函数使用均方差，同时采用 logistic 的 sigmoid 激活函数，而组合函数是求和函数，则采用链式法则求解参数的更新可以写作：

$$\begin{aligned}\frac{\partial J(w)}{\partial w}&=2\left(f(x'w+b)-y\right)\frac{\partial f(x'w+b)}{\partial w}\\&=2\left(f(x'w+b)-y\right)f(x'w+b)\left[1-f(x'w+b)\right]x\end{aligned}$$

将上述公式带入前面提到的梯度递减算法的参数更新步骤中，就可以得到新的参数估计。更新偏置项 b 采用几乎一样的公式，只是这个时候 $x=1$，因此上面公式中最后的 x 就消掉了。

3.5 后向传播算法

上一节对梯度递减算法进行了介绍，如果这个神经网络只有一层，那么反复运用这个算法到损失函数，依照上面公式更新参数直到收敛就好了。如果神经网络模型是一个深度模型，在输入层和输出层之间包含很多隐含层的话，就需要一个高效率的算法来尽量减少计算量。后向传播算法（Backpropagation）就是一种为了快速估计深度神经网络中的权重值而设计的算法。

设定 $f^0,\cdots,f^N$ 代表 $1,\cdots,N$ 层的决策函数，其中 0 对应于输入层，而 N 对应于输出层。如果已知各层的权重值和偏置项估计值，那么可以采用下面的递归算法快速求得在当前参数值下的损失函数大小：

$$
\begin{aligned}
&f^0 = x \\
&f^1 = g(w^0 h^0 + b^0) \\
&f^2 = g(w^1 h^1 + b^1) \\
&\cdots\cdots \\
&f^{N-1} = g(w^{N-2} h^{N-2} + b^{N-2}) \\
&f^N = g(w^{N-1} h^{N-1} + b^{N-1})
\end{aligned}
$$

为了更新参数值，即权重值和偏置项的估值，后向传播算法先正向计算组合函数和其他相关数值，再反向从输出层 N 求解损失函数开始，按照梯度递减算法逐次往输入层回算参数的更新量。

(1) 按照给定的参数从输入层到输出层正向计算组合函数 h 的取值

(2) $\delta^N = 2(f^N - y)\frac{\partial}{\partial w} g(w^{N-1} h^{N-1} + b^{N-1})$

(3) $\delta^{N-1} = (w^{N-1}\Delta w^N)\frac{\partial}{\partial w} g(w^{N-2} h^{N-2} + b^{N-2})$

(4) $\Delta w^{N-1} = \delta^N h^{N-1}$

(5) $\Delta b^{N-1} = \delta^{N-1}$

(6) 对 $N-1, \cdots, 1$ 层重复以上步骤

在深度学习模型所需的计算中会大量使用链式法则，这就会使很多计算结果得到重复使用，后向传播算法将这些中间结果保存下来可以极大地减少计算量，提高模型拟合速度。因为在每一层都使用同样的函数：$f : \mathbb{R} \to \mathbb{R}$，在这些层中，有：$f^{(1)} = f(w), f^{(2)} = f(f^{(1)}), f^{(3)} = f(f^{(2)})$，其中上标代表对应的网络层，要计算 $\dfrac{\partial f^{(3)}}{\partial w}$，可以通过链式法则得到：

$$
\begin{aligned}
&\frac{\partial f^{(3)}}{\partial w} \\
&= \frac{f^{(3)}}{\partial f^{(2)}} \frac{\partial f^{(2)}}{\partial f^{(1)}} \frac{\partial f^{(1)}}{\partial w} \\
&= f'(f^{(2)}) f'(f^{(1)}) f'(w) \\
&= f'(f(f(w)))) f'(f(w)) f'(w)
\end{aligned}
$$

可以看到，只需计算 $f(w)$ 一次，保存在变量 $f^{(1)}$ 中，就可以在以后的计算中使用多次，层数越多，效果越明显。相反，如果不是反向求解参数更新量，而是在正向传播那一步求解参数更新量，那么每一步中的 $f(w)$ 都要重新求解，计算量大增。可以说，后向传播算法是神经网络模型普及的基础之一。

4 Keras 入门

4.1 Keras 简介

在现在的深度学习软件中，我们选择 Keras 来介绍，并在以后的几章中运用 Keras 来解决实际问题。Keras 这个名字源于希腊古典史诗《奥德赛》里的牛角之门（Gate of Horn），是真实事物进出梦境和现实的地方。《奥德赛》里面说：象牙之门（Gate Of Ivory）内只是一场无法应验的梦境，唯有走进牛角之门（Gate Of Horn）奋斗的人，才能拥有真正的回报（“Those that come through the Ivory Gate cheat us with empty promises that never see fulfillment. Those that come through the Gate of Horn inform the dreamer of the truth”）（见图4.1）。Keras 作者的寓意不可谓不深刻。



图 4.1　象牙之门与牛角之门（图片来自于 http://www.coryianshaferlpc.com/）

Keras 是 Python 中一个以 CNTK、TensorFlow 或者 Theano 为计算后台的深度学习建模环境。相对于常见的几种深度学习计算软件，比如 TensorFlow、Theano、Caffe、CNTK、Torch 等，Keras 在实际应用中有如下几个显著的优点。

- Keras 在设计时以人为本，强调快速建模，用户能快速地将所需模型的结构映射到 Keras 代码中，尽可能减少编写代码的工作量，特别是对于成熟的模型类型，从而加快开发速度。
- 支持现有的常见结构，比如卷积神经网络、时间递归神经网络等，足以应对大量的常见应用场景。
- 高度模块化，用户几乎能够任意组合各个模块来构造所需的模型。在 Keras 中，任何神经网络模型都可以被描述为一个图模型或者序列模型，其中的部件被划分为以下模块：神经网络层、损失函数、激活函数、初始化方法、正则化方法、优化引擎。这些模块可以以任意合理地方式放入图模型或者序列模型中来构造所需的模型，用户并不需要知道每个模块后面的细节。这种方式相比其他软件需要用户编写大量代码或者用特定语言来描述神经网络结构的方法效率高很多，也不容易出错。
- 基于 Python，用户也可以使用 Python 代码来描述模型，因此易用性、可扩展性都非常高。用户可以非常容易地编写自己的定制模块，或者对已有模块进行修改或者扩展，因此可以非常方便地开发和应用新的模型与方法，加快迭代速度。
- 能在 CPU 和 GPU 之间无缝切换，适用于不同的应用环境。当然，我们强烈推荐 GPU 环境。

4.2 Keras 中的数据处理

我们首先介绍 Keras 中的数据处理。任何机器学习软件对所需数据的规范和格式都有自己的要求，在深度学习中通常要求输入算法的数据为一个多维矩阵的形式，而用于建模的数据来源多种多样，因此在进行任何机器学习之前都要对原始数据按照软件规定进行处理。

Keras 针对几种常见的输入深度学习模型和输入数据形态，提供了几个易于使用的工具来处理数据，包括针对序列模型的数据预处理、针对文字输入的数据处理，以及针对图片输入的数据处理。所有函数都在 Keras.preprocessing 这个库里面，分别有 text、sequence 和 image 三个子库。

4.2.1 文字预处理

在文字的建模实践中，一般都需要把原始文字拆解成单字、单词或者词组，然后将这些拆分后的要素进行索引、标记化供机器学习算法使用。这种预处理叫作标注（Tokenize）。虽然这些功能都可以使用 Python 来实现，但是 Keras 提供了现成的方法，既方便，又高效。一般来说，对于已经读入的文字的预处理包含以下几个步骤。

(1) 文字拆分。

(2) 建立索引。

(3) 序列补齐（Padding）。

(4) 转换为矩阵。

(5) 使用标注类批量处理文本文件。

所有跟文字相关的预处理函数都在 Keras.preprocessing.text 这个子库里。但是这是为英文文字设计的，如果是处理中文，因为中英文的差异，建议使用结巴分词里提供的切分函数 cut 来进行文字拆分。

文字拆分

文字拆分是第一步，这时候就需要用到库里的 text_to_word_sequence 函数，顾名思义，就是将一段文字根据预定义的分隔符（不能为空值）切分成字符串或者单词（英文）。这个函数返回一个单词列表，但是会先处理一下，比如将过滤表中的字符过滤掉，或者将字符都变为小写字母等。下面来看几个例子。首先是一个英文例子，我们就用上面提到的《奥德赛》里面讲象牙之门和牛角之门的那段话为例。

```
txt ="Those that come through the Ivory Gate cheat us with empty promises
that never see fulfillment. Those that come through the Gate of Horn inform
the dreamer of the truth"
out1 = text_to_word_sequence(txt)
print(out1[:6])

out2 = text_to_word_sequence(txt, lower=False)
print(out2[:6])

out3 = text_to_word_sequence(txt, lower=False, filters="Tha")
print(out3[:6])
```

输出结果分别是：

```
['those', 'that', 'come', 'through', 'the', 'ivory']
['Those', 'that', 'come', 'through', 'the', 'Ivory']
['ose', 't', 't', 'come', 't', 'roug']
```

那么在中文中直接使用这个函数会有怎样的效果呢？我们看看下面的借用《红楼梦》中第一段话的例子。

```
chn='此开卷第一回也。作者自云：因曾历过一番梦幻之后，故将真事隐去，而借"通灵"之说，撰此《石头记》一书也。故曰"甄士隐"云云. 但书中所记何事何人？自又云："今风尘碌碌，一事无成，忽念及当日所有之女子，一一细考较去，觉其行止见识，皆出于我之上. 何我堂堂须眉，诚不若彼裙钗哉？实愧则有余，悔又无益之大无可如何之日也！当此，则自欲将已往所赖天恩祖德，锦衣纨绔之时，饫甘餍肥之日，背父兄教育之恩，负师友规谈之德，以至今日一技无成，半生潦倒之罪，编述一集，以告天下人：我之罪固不免，然闺阁中本自历历有人，万不可因我之不肖，自护己短，一并使其泯灭也. 虽今日之茅椽蓬牖，瓦灶绳床，其晨夕风露，阶柳庭花，亦未有妨我之襟怀笔墨者. 虽我未学，下笔无文，又何妨用假语村言，敷演出一段故事来，亦可使闺阁昭传，复可悦世之目，破人愁闷，不亦宜乎？故曰"贾雨村"云云。'

chout1 = text_to_word_sequence(chn)
print(len(chout1), chout1[:2])

chout2 = text_to_word_sequence(chn, lower=True)
print(len(chout2), chout2[:2])

chout3 = text_to_word_sequence(chn, lower=True, filters="。：")
print(len(chout3), chout3[:4])
```

结果很有趣：

```
8 ['此开卷第一回也。作者自云：因曾历过一番梦幻之后，故将真事隐去，而借', '通灵']
8 ['此开卷第一回也。作者自云：因曾历过一番梦幻之后，故将真事隐去，而借', '通灵']
6 ['此开卷第一回也', '作者自云', '因曾历过一番梦幻之后，故将真事隐去，而借"通灵"之说，撰此《石头记》一书也', '故曰"甄士隐"云云. 但书中所记何事何人？自又云']
```

我们看到，在默认情况下，text_to_word_sequence 函数使用引号作为分隔符，在“通灵”一词的引号前后将句子切分。因为中文没有大小写一说，因此 lower 选项没有任何作用。但是过滤选项的表现很奇怪。在第三个命令中，使用了 filters="。：" 选项，这时分隔符发生了变化，这个函数不再使用引号作为分隔符了，而是使用过滤符号作为分隔符。显然对于中文应该使用专门为中文设计的切分工具，我们选用“结巴分词”（jieba）作为切分工具。“结巴分词”是一个基于 Python 的中文分词组件，可以通过 pip install jieba 来自动安装（如果 Python 环境是 Python 3，请使用 pip3 来安装）。

根据结巴分词的介绍，其使用了如下算法来进行中文分词。

- 基于前缀词典实现高效的词图扫描，生成句子中汉字所有可能成词情况所构成的有向无环图（DAG）。
- 采用动态规划查找最大概率路径，找出基于词频的最大切分组合。
- 对于未登录词，采用了基于汉字成词能力的 HMM 模型，使用 Viterbi 算法。

对于中文分词，结巴分词提供了 jieba.cut 和 jieba.cut_for_search 函数，其中 cut 是最常用的，cut_for_search 是为搜索引擎构造索引所采用的比精确分词模式颗粒度略细的分词方法，返回一个可迭代的生成器（Generator）对象，可以使用 for 循环来获取分割后的单词。它们各自对应一个返回列表的函数，分别是 lcut 和 lcut_for_search，其用法一样，只是返回数据类型不同。这里主要讲解 cut 函数，但是为了演示方便，在示例中使用返回数据类型为列表的 lcut 函数。

cut/lcut 接受三个参数，分别是需要分割的 Unicode 字符串、分词是否采用细颗粒度模式和是否使用 HMM 模型。还是用上面的例子来演示。

```
chnout4 = jieba.lcut(chn, cut_all=False)
print(len(chnout4), chnout4[:8])

chnout5 = jieba.lcut(chn, cut_all=True)
print(len(chnout5), chnout5[:15])

chnout6 = jieba.lcut(chn, cut_all=True, HMM=True)
print(len(chnout6), chnout6[:15])
```

其结果如下：

```
233 ['此', '开卷', '第一回', '也', '。', '作者', '自云', '：']
345 ['此', '开卷', '第一', '第一回', '一回', '也', '', '', '作者', '自', '云
', '', '', '因', '曾']
```

```
345 ['此', '开卷', '第一', '第一回', '一回', '也', '', '', '作者', '自', '云
', '', '', '因', '曾']
```

我们看到，当使用 cut_all=True 选项时，返回的单词颗粒度较细，“第一回”被拆分成三个可能的单词：“第一”“第一回”“一回”，同时标点符号在返回列表中消失了，只是一个空的元素。在这个比较简单的例子中，使用 HMM 模型不影响返回的结果。我们将这个练习留给读者。

建立索引

完成分词以后，得到的单字或者单词并不能直接用于建模，还需要将它们转换成数字序号，才能进行后续处理。这就是建立索引。建立索引的方法很简单，对于拆分出来的每一个单字或者单词，排序之后编号即可。下面的代码是 Keras 的预处理模块中用于建立索引的方法，略有简化。假设我们已经有了一个字符串列表，比如上面通过 text_to_word_sequence 生成的 out1 变量，那么可以通过下面的代码来生成拆分后单字或者单词的索引。

```
out1.sort(reverse=True)
dict(list(zip(out1, np.arange(len(out1)))))
```

在上面的代码中，第一句是对原有字符串反向排序；第二句有三个动作，首先通过 zip 命令将每个单词依次与序号配对，然后通过 list 命令将配对的数据改为列表，每个元素是诸如 ('with', 0) 这样的一对，最后应用字典命令将列表修改为字典即可完成索引。

建立索引也可以使用 One Hot 编码法，即对于 K 个不同的单字或者单词，依次设定一个 1 到 K 之间的数值来索引这 K 个单字或者单词构成的词汇表。这可以很容易地使用 one_hot 函数来实现。

这个函数有两个参数：一个是待索引的字符串列表；一个是最大索引值 n。这个函数将输入的字符串列表按照规则将其分配给 $0, \cdots, n-1$ 共 n 个索引值其中之一。那么这个规则是怎样的呢？我们看下面的例子。

```
xin = [0,1,2,3,4,5,6,7,8,9,10,11,12,13]
tout=(text_to_word_sequence(str(xin)))
xout=one_hot(str(xin), 5)
for s in range(len(xin)):
    print(s, hash(tout[s])%(5-1), xout[s])
```

结果是这样的：

```
0 2 3
1 1 4
2 1 2
3 3 2
4 1 2
5 0 1
6 2 3
7 2 3
8 3 2
9 0 1
10 3 2
11 1 4
12 0 1
13 2 3
```

一般要将大量不同的数据映射到一个有限空间中，通常采用的方法都是哈希表，one_hot 函数也不例外。我们发现这个函数其实是按照列表输出元素的哈希值取模作为输出的。如果输入的是整数值，那么哈希值就等于这个整数值，而对于连续的字符串，比如一段文字，该函数先使用 text_to_word_sequence 函数将其切分。因此，如果输入的是中文长字符串，则必须先用结巴分词切分以后再使用该函数建立索引。该函数的源程序如下：

```
def one_hot(text, n, filters=base_filter(), lower=True, split=" "):
    seq = text_to_word_sequence(text, filters=filters, lower=lower, split=
    split)
    return [(abs(hash(w)) % (n - 1) + 1) for w in seq]
```

如果最大索引值设定为不同的单字或者单词的数量，则能实现跟上面一样的索引目的。

这个方法的问题在于如果最大索引值不小心设置成小于不同单字或者单词的数量，就会有哈希碰撞的问题，那么取模后同样取值的字符实质上并不具备任何关系。而如果最大索引值跟不同字符串的数量一样的话，就会产生极度稀疏的输入矩阵。无论哪种情况，这么建立的索引在以后建模时效果都不好。

序列补齐

最终索引之后的文字信息会被按照索引编号放入多维矩阵中用来建模。这个多维矩阵的行宽对应于所有拆分后的单字或者单词，但是在将索引放入矩阵中之前，需要先进行序列补齐的工作。这是因为将一段话拆分成单一的词以后，丢失了重要的上下文信息，因此将上下文的一组词放在一起建模能保持原来的上下文信息，从而提高建模的质量。序列补齐分两种情况。

第一种情况是自然的文本序列，比如微博或者推特上的一段话，都是一个自然的单字或者单词序列，而待建模的数据是由很多微博或者推特组成的，或者对一组文章进行建模，每篇文章中的每一句话构成一个文本序列。这个时候每句话的长度不一，需要进行补齐为统一长度。

第二种情况是将一个由 K 个（K 较大）具备一定顺序的单词串拆分成小块的连续子串，每个子串只有 M 个（$M < K$）单词。这种情况一般是一大段文字按照固定长度移动一个窗口，将窗口内的单词索引载入多维矩阵的每一行，因此一句话可能会对应于矩阵的多行数据，形成时间步（timestep）。

对于序列补齐，可以使用 pad_sequences 函数，其输入的要素是列表串（list of list）。假设有一个列表串，包含了单词的索引号，下面的程序展示了在不同设置选项下使用这个函数如何补齐序列。

```
from Keras.preprocessing.sequence import pad_sequences
x = [[1,2,3], [4,5], [6,7,8,9]]
y0 = pad_sequences(x)
y1 = pad_sequences(x, maxlen=5, padding='post')
y2 = pad_sequences(x, maxlen=3, padding='post')
y3 = pad_sequences(x, maxlen=3, padding='pre')
print(y0)
print("=====")
print(y1)
print("=====")
print(y2)
print("=====")
print(y3)
```

其结果如下：

```
[[0 1 2 3]
 [0 0 4 5]
```

```
 [6 7 8 9]]
=====
[[1 2 3 0 0]
 [4 5 0 0 0]
 [6 7 8 9 0]]
=====
[[1 2 3]
 [4 5 0]
 [7 8 9]]
=====
[[1 2 3]
 [0 4 5]
 [7 8 9]]
```

我们看到，其中的 padding 选项指定是从后面补齐还是从前面补齐，补齐的索引数字默认为 0，不过可以通过 value 选项修改（该选项在上例中没有明确标识）。如果不用 maxlen 选项设定补齐序列的长度，则按照最长列表元素的长度来设定。如果设定的补齐序列的长度小于一些列表元素的长度，那么会产生截断。截断的标准是假如补齐序列的长度为 k，则保留最后 k 个索引值。

转换为矩阵

所有的建模都只能使用多维矩阵，因此最后必须将索引过的文字元素转换成可以用于建模的矩阵。Keras 提供了两种方法。第一种方法是使用 pad_sequences 函数。上面我们看到，对于一个已经建立了索引的文本句子列表集合，这个函数可以生成一个宽度为指定句子长度、高度为句子个数的矩阵。假设 text 是一个包含每一句话的列表，而每一句话已经通过 text_to_word_sequence 或者 jieba.cut 函数拆分为单字或者单词的列表，那么下面的程序通过将文字映射到其对应的索引上，再通过序列补齐函数建立相应的矩阵。

```
max_sentence_len=50
X = []
for sentences in text:
        x = [word_idx[w] for w in sentences]
        X.append(x)

pad_sequences(X, maxlen=max_sentence_len)
```

第二种方法是使用下面将要介绍的标注类来进行。因为一般要将文字转换为矩阵的情况多是对应于多个不同的文本（比如不同的小说），或者同一个文本的不同段落（比如同一个小说的不同章节等），因此很自然的是对应于大量元素的列表串。在这种情况下，标注类提供了一系列整合的方法来对文本按照上述步骤进行处理。

使用标注类批量处理文本文件

当批量处理文本文件时，需要一种更高效的方法。Keras 提供了一个标注类（Tokenizer class）来进行文本处理。当批量处理文本文件时，一般所有文本会被读入一个大的列表中，每一个元素是单个文件的文本或者一大段文本。上述方法都是针对单一字符串设计的，而标注类中的方法是针对一个文本列表设计的。标注类中包含了几个非常实用的方法，也能返回所生成的数据的重要统计量，方便后续建模。这个类对应的操作数据有两种类型，分别是文本列表和单词串列表，对应的方法包含“texts”或者“sequences”字样，对应于文本列表的方法都是将文本拆分成单词串以后执行相应的操作。下面举一个简单的例子。

假设我们已经通过 open(file).read() 函数将一系列文本文件读入 alltext 这个列表变量中，每一个元素是一个文本文件中的文本。在进行所有预处理之前，我们先初始化标注对象：

```
from Keras.preprocessing.text import Tokenizer
tokenizer = Tokenizer(nb_words=1000)
tokenizer.fit_on_text(alltext)
```

fit_on_text() 函数的作用是对于输入的文本计算一些关键统计量，并对里面的元素进行索引。

- 首先，依次遍历文本列表变量元素，对于每一个字符串元素，使用上面提到的 text_to_word_sequence 函数进行拆分，并统一为小写字符。
- 其次，计算单词出现的总频率和在不同文件中分别出现的频率，并对单词表排序。
- 最后，计算总的单词量，并对每一个单词建立一个总的索引和一个在不同文件中的索引。

完成上述准备工作之后，就可以对整个列表中的元素进行拆分了。

```
word_sequences = tokenizer.texts_to_sequences(alltext)
```

将每一个文本字符串拆分成单词以后，还需要对每个字符串做序列补齐的工作，才能将其最终转换为可用于建模的矩阵。这时就要用到上面提到的 pad_sequences 函数，其用法一样：

```
padded_word_sequences = pad_sequences(word-sequence, maxlen=
MAX_sequence_length);
```

在标注类中有两个方法是用来将文本序列列表转换为待建模矩阵的，即 text_to_matrix 和 sequence_to_matrix。其中 text_to_matrix 基于后者，对从文本序列列表中抽取的每一个序列元素应用 sequence_to_matrix 转换为矩阵。

4.2.2 序列数据预处理

关于对序列数据的处理，我们在上一节中已经提到了一个单词串补齐的例子。但是序列数据不光有字符串，还有时间序列数据，不过其处理行为跟上述例子是一样的，这里不再详解。其实不论是补齐还是截断，其操作都是将相邻的连续 N 个元素连在一起，即跟自然语言处理中的 N 元语法（N-Gram）模型类似。

与此相似，但又不同的另外一种对序列数据的处理方法叫作跳跃语法（Skip Gram）模型。这是 Tomas Mikolov 在 2013 年提出的单词表述（Word Representation）模型，它把每个单词映射到一个 M 维的空间，它有一个更著名的别名，即 Word2Vec。这个模型虽然是处理序列数据的，不过并没有考虑词的次序，而是单纯的一个从单词到向量的映射模型。在 Keras 的预处理模块中有一个 skipgrams 的函数，将一个词向量索引标号按照两种可选方式转化为一系列两两元素的组合（$w1, w2$）和标注 z。如果 $w2$ 跟 $w1$ 是紧挨着的，则标注 z 为 1，为正样本；如果 $w2$ 是从不相邻的其他元素中随机抽取的，则标注 z 为负样本。

下面看一个例子。

```
z0 = skipgrams([1,2,3],3)
res=list(zip(z0[0], z0[1]))
for s in res:
    print(s)
```

输出结果为：

```
([3, 2], 0)
((1, 1], 0)
```

```
([2, 2], 0)
([3, 2], 0)
([2, 3], 1)
([2, 2], 0)
([1, 3], 1)
([2, 1], 1)
([1, 1], 0)
([3, 2], 1)
([3, 1], 1)
([1, 2], 1)
```

4.2.3 图片数据输入

Keras 为图片数据的输入提供了一个很好的接口，即 Keras.preprocessing.image.ImageDataGenerator 类。这个类生成一个数据生成器（Generator）对象，依照循环批量产生对应于图像信息的多维矩阵。根据后台运行环境的不同，比如是 TensorFlow 还是 Theano，多维矩阵的不同维度对应的信息分别是图像二维的像素点，第三维对应于色彩通道，因此如果是灰度图像，那么色彩通道只有一个维度；如果是 RGB 色彩，那么色彩通道有三个维度。

4.3 Keras 中的模型

在 Keras 中设定了两类深度学习模型：一类是序列模型（Sequential 类）；一类是通用模型（Model 类）。其差异在于不同的拓扑结构。

序列模型

序列模型属于通用模型的一个子类，因为很常见，所以这里单独列出来进行介绍。这种模型各层之间是依次顺序的线性关系，在第 k 层和第 $k+1$ 层之间可以加上各种元素来构造神经网络。这些元素可以通过一个列表来制定，然后作为参数传递给序列模型来生成相应的模型。示例代码如下：

```
1 from Keras.models import Sequential
2 from Keras.layers import Dense, Activation
3
```

```
layers = [ Dense(32, input_shape=(784,)),
                Activation('relu'),
                Dense(10),
                Activation('softmax')]
model = Sequential(layers)
```

除一开始直接在一个列表中指定所有元素外，也可以像下面这个例子一样逐层添加：

```
from Keras.models import Sequential
from Keras.layers import Dense, Activation

model = Sequential()
model.add(Dense(32, input_shape=(784,)) )
model.add(Activation('relu'))
model.add(Dense(10))
model.add(Activation('softmax') )
```

通用模型

通用模型可以用来设计非常复杂、任意拓扑结构的神经网络，例如有向无环网络、共享层网络等。类似于序列模型，通用模型通过函数化的应用接口来定义模型。使用函数化的应用接口有多个好处，比如：决定函数执行结果的唯一要素是其返回值，而决定返回值的唯一要素则是其参数，这大大减轻了代码测试的工作量；因为函数式语言是一个形式系统，只要能用数学运算表达的就能用这种语言来表述，因此，只要在数学上是等价的，那么机器就可以使用等价的但是效率更高的代码来代替效率低的代码而不影响结果。这一方面方便了分析师写程序；另一方面又从数学上保证了代码效率，实现了人工时间和机器时间的双重高效。有兴趣的读者可以去阅读一些函数式编程的书来帮助理解。

在函数式编程中，操作对象都是函数，函数也作为参数来传递，因此可以很方便地转化为一个函数接口供其他函数调用，比如有一个计算任意两个实数乘积的函数：double times(double x, double y)，那么 Triple = double times(double x, 3) 就定义了一个计算 3 倍数的新函数，只需要一个参数，从程序的角度来看会继续调用 times 函数，并把第二个参数设置为 3，创建一个 times 函数的封装函数。

在通用模型中，使用同样的方法来定义模型的要素和结构。在定义的时候，从输入的多维矩阵开始，然后定义各层及其要素，最后定义输出层。将输入层和输出层作为参数纳入通用模型中就可以定义一个模型对象，并进行编译和拟合。下面的例子来自于

Keras 手册，用一个全连接神经网络拟合一个手写阿拉伯数字的分类模型。输入的数据是 28×28 的图像。

首先，载入相关模块。

```
from Keras.layers import Input, Dense
from Keras.models import Model
```

然后，定义输入层 Input，主要是为了定义输入的多维矩阵的尺寸。在这里因为每一个图像都被拉平为 784 个像素点的向量，因此这个多维矩阵的尺寸为（784，）的向量。

```
input = Input(shape = (784, ) )
```

现在定义各个连接层，包括相应的激活函数。假设从输入层开始，定义两个隐含层，都有 64 个神经元，都使用 relu 激活函数。

```
x = Dense(64, activation='relu')(inputs)
x = Dense(64, activation='relu')(x)
```

第一个隐含层使用输入层作为参数，而第二个隐含层使用第一个隐含层作为参数，这跟上面的封装例子类似，体现了函数式编程的优点。

接下来定义输出层，使用最近的隐含层作为参数。

```
y = Dense(10, activation='softmax')(x)
```

所有要素都齐备以后，就可以定义模型对象了，参数很简单，分别是输入和输出，其中输出包含了中间的各种信息。

```
model = Model(inputs = input, outputs = y);
```

最后，当模型对象定义完成以后，就可以进行编译了，并对数据进行拟合。拟合的时候也有两个参数，分别对应于输入和输出。

```
model.compile(optimizer='rmsprop',  loss='categorical_crossentropy',
metrics=['accuracy'])
model.fit(data, labels)
```

我们看到，对于序列模型和通用模型，它们的主要差异在于如何定义从输入层到输出层的各层结构。

- 首先，在序列模型里，是先定义序列模型对象的；而在通用模型中是先定义从输入层到输出层各层要素的，包括尺寸结构。

- 其次，在序列模型中，当有了一个模型对象以后，可以通过 add 方法对其依次添加各层信息，包括激活函数和网络尺寸来定义整个神经网络；而在通用模型中，是通过不停地封装含有各层网络结构的函数作为参数来定义网络结构的。
- 最后，在序列模型中，各层只能依次线性添加；而在通用模型中，因为采用了封装的概念，可以在原有的网络结构上应用新的结构来快速生成新的模型，因此灵活度要高很多，特别是在具有多种类型的输入数据的情况下，比如在 Keras 手册中就举了一个教神经网络看视频进行自然语言问答的例子。在这个例子中，输入数据有两种：一是视频图像；二是自然语言的提问。首先通过构造多层卷积神经网络使用序列模型来对图像编码，然后将这个模型放入 TimeDistributed 函数中建立视频编码，最后使用 LSTM 对编码建模，同时对自然语言也进行从文字到向量的转换，在合并两个网络以后，将合并的网络作为参数输入下一个全连接层进行计算，并输出可能的回答。

虽然对于大部分工作，序列模型已经能够有效应对，但是函数式接口的通用模型为分析师提供了更强大的工具。

4.4 Keras 中的重要对象

Keras 预先定义了很多对象用于帮助构造 Keras 的网络结构，比如常用的激活函数、参数初始化方法、正则化方法等。这些丰富的预定义对象是让 Keras 方便易用的重要前提条件。下面简要介绍常用的激活对象、初始化对象和正则化对象。

激活对象

在定义网络层时，使用什么激活函数是很重要的选择。Keras 提供了大量预定义好的激活函数，方便定制各种不同的网络结构。在 Keras 中使用激活对象有两种方法：一是单独定义一个激活层；二是在前置层里面通过激活选项来定义所需的激活函数。比如，下面两段代码是等效的，前一段是通过激活层来使用激活对象的；后一段是使用前置层的激活选项来使用激活对象的。

```
model.add(Dense(64, input_shape=(784,)))
model.add(Activation('tanh'))
```

```
model.add(Dense(64, input_shape=(784, ), activation='tanh') )
```

Keras 预定义的激活函数可以通过预先定义好的字符串来引用，比如上面代码使用了 tanh 激活函数。下面简要介绍这些预定义的激活函数。

- **softmax**：这个激活函数也被称为归一化的指数函数，是逻辑函数的扩展，能将 K 维的实数域上的数值压缩到 K 维的（0, 1）值域上，并且 K 个数值的和为 1。这个函数可以写作：

$$s(x)_j = \frac{\exp(x_j)}{\sum_i^K \exp(x_i)}, \quad j = 1, \cdots, K$$

 在概率理论中，这个公式描述了一个有 K 种不同取值的离散变量的分布，因此也很自然地出现在其他多类别分类算法中，比如多类别逻辑回归、多类别线性分类器等都使用这个函数。

- **softplus**：这个激活函数将原始值从任意实数区间投影到正实数区间，即值域从整个实数域变为 $(0, \text{inf})$。用公式表示如下：

$$s(x) = \ln(1 + \exp(x))$$

- **softsign**：这个激活函数起到的作用类似于三角函数，将实数域上的数值投影到 $(-1, 1)$ 区间。用公式表示如下：

$$s(x) = \frac{x}{1 + \|x\|}$$

- **elu**：这个激活函数的英文全称为 Exponential Linear Unit，带一个参数 α。这个激活函数用公式表示为：

$$s(x) = x < 0 ? \alpha(\exp(x) - 1) : x$$

 即当参数小于 0 时，使用 $\alpha(\exp(x) - 1)$ 作为输出，如果参数大于或等于 0，则输出参数的值，因此其值域为 $(-\alpha, \text{inf})$。

- **relu**：这个激活函数的英文全称为 Rectified Linear Unit，是一个阶梯函数，当参数小于 0 时，其取值为 0；当参数大于或等于 0 时，则保持参数的值，因此将实数域上的数值投影到 $[0, \text{inf})$ 区间。

- **tanh**：这个激活函数运用三角函数中的双曲正切函数将实数域上的取值压缩到 $(-1, 1)$ 区间。用公式表示如下：

$$s(x) = \tanh(x) = \frac{2}{1+\exp(-2x)} - 1$$

- **sigmoid**：这个激活函数在 Keras 中特指逻辑函数，是一个将实数域上的取值压缩到 $(0,1)$ 区间的函数。如果读者有统计学背景，那么对这个函数应该非常熟悉。用公式表示如下：

$$s(x) = \frac{1}{1+\exp(-x)}$$

 sigmoid 指代其压缩后的取值具有 S 形的曲线。有时候 tanh 也被纳入 sigmoid 这类函数中。

- **hard_sigmoid**：这是上面提到的标准 sigmoid 激活函数的多段线性逼近形式，旨在避免 exp() 函数的计算，加快速度。该函数可以用如下公式表示：

$$s(x) = \begin{cases} 0 & ,x < -2.5 \\ 0.2 * x + 0.5 & ,-2.5 \leqslant x \leqslant 2.5 \\ 1 & ,x > 2.5 \end{cases}$$

- **linear**：线性激活函数不对参数做任何变换，即 $f(x) = x$。当激活选项设置为 None 时，即选择线性激活函数。

初始化对象

初始化对象（Initializer）用于随机设定网络层激活函数中的权重值或者偏置项的初始值，包括 kernel_initializer 和 bias_initializer。好的权重初始化值能帮助加快模型收敛速度。Keras 预先定义了很多不同的初始化对象，包括：

- Zeros，将所有参数值都初始化为 0。
- Ones，将所有参数值都初始化为 1。
- Constant(value=1)，将所有参数值都初始化为某一个常量，比如这里设置为 1。
- RandomNormal，将所有参数值都按照一个正态分布所生成的随机数来初始化。正态分布的均值默认为 0，而标准差默认为 0.05。可以通过 mean 和 stddev 选项来修改。
- TruncatedNormal，使用一个截断正态分布生成的随机数来初始化参数向量，默认参数均值为 0，标准差为 0.05。对于均值的两个标准差之外的随机数会被遗弃并重新取样。这种初始化方法既有一定的多样性，又不会产生特别偏的值，因此是比较推荐的方法。针对不同的常用分布选项，Keras 还提供了两个基于

这种方法的特例，即 glorot_normal 和 he_normal。前者的标准差不再是 0.05，而是输入向量和输出向量的维度的函数：stddev $= \sqrt{2/(n_1+n_2)}$，其中 n_1 是输入向量的维度，而 n_2 是输出向量的维度；后者的标准差只是输入向量的维度的函数：stddev $= \sqrt{2/n_1}$。

- RandomUniform，按照均匀分布所生成的随机数来初始化参数值，默认的分布参数最小值为 -0.05，最大值为 0.05，可以通过 minval 和 maxval 选项分别修改。针对常用的分布选项，Keras 还提供了两个基于这个分布的特例即 glorot_uniform 和 he_uniform。前者均匀分布的上下限是输入向量和输出向量的维度的函数：minval/maxval $= -/+\sqrt{6/(n_1+n_2)}$；而在后者上下限只是输入向量的维度的函数：minval/maxval $= -/+\sqrt{6/n_1}$。
- 自定义，用户可以自定义一个与参数维度相符合的初始化函数。下面的例子来自于 Keras 手册，使用后台的正态分布函数生成一组初始值，在定义网络层的时候调用这个函数即可。

```
from Keras import backend as K

def my_init(shape, dtype=None):
    return K.random_normal(shape, dtype=dtype)

model.add(Dense(64, kernel_initializer=my_init))
```

正则化对象

在建模的时候，正则化是防止过度拟合的一个很常用的手段。在神经网络中也提供了正则化的手段，分别应用于权重参数、偏置项以及激活函数，对应的选项分别是 kernel_regularizer、bias_reuglarizier 和 activity_regularizer。它们都可以应用 Keras.regularizier.Regularizer 对象，这个对象提供了定义好的一阶、二阶和混合的正则化方法，分别将前面的 Regularizier 替换为 l1(x)、l2(x) 和 l1_l2(x1, x2)，其中 x 或者 x1,x2 为非负实数，表明正则化的权重。

读者也可以设计自己的针对权重矩阵的正则项，只要接受权重矩阵为参数，并且输出单个数值即可。Keras 手册提供的例子如下：

```
from Keras import backend as K

def l1_reg(weight_matrix):
```

```
    return 0.01 * K.sum(K.abs(weight_matrix))

model.add(Dense(64, input_dim=64,  kernel_regularizer=l1_reg)
```

在这个例子中，用户自己定义了一个比例为 0.01 的一阶正则化项，返回的单个数值是权重参数的绝对值的和，乘以 0.01 这个比例，其用法跟预先提供的 regularizier.l1(x) 对象是一样的。

4.5 Keras 中的网络层构造

从上面的介绍看到，在 Keras 中，定义神经网络的具体结构是通过组织不同的网络层（Layer）来实现的。因此了解各种网络层的作用还是很有必要的。

核心层

核心层（Core Layer）是构成神经网络最常用的网络层的集合，包括：全连接层、激活层、放弃层、扁平化层、重构层、排列层、向量反复层、Lambda 层、激活值正则化层、掩盖层。所有的层都包含一个输入端和一个输出端，中间包含激活函数以及其他相关参数等。

(1) 全连接层。在神经网络中最常见的网络层就是全连接层，在这个层中实现对神经网络里面的神经元的激活。比如：$y = g(x'w + b)$，其中 w 是该层的权重向量，b 是偏置项，$g()$ 是激活函数。如果 use_bias 选项设置为 False，那么偏置项为 0。常见的引用全连接层的语句如下：

```
model.add(Dense(32, activation='relu', use_bias=True, kernel_initializer='
uniform', kernel_initializer='uniform', activity_regularizer=regularizers.
l1_l2(0.2, 0.5) )
```

在上面的语句中：

- 32，表示向下一层输出向量的维度，
- activation='relu'，表示使用 relu 函数作为对应神经元的激活函数。
- kernel_initializer='uniform'，表示使用均匀分布来初始化权重向量，类似的选项也可以用在偏置项上。读者可以参考前面的“初始化对象”部分的介绍。
- activity_regularizer=regularizers.l1_l2(0.2, 0.5)，表示使用弹性网作为正则项，其中一阶的正则化参数为 0.2，二阶的正则化参数为 0.5。

(2) 激活层。激活层是对上一层的输出应用激活函数的网络层，这是除应用 activation 选项之外，另一种指定激活函数的方式。其用法很简单，只要在参数中指明所需的激活函数即可，预先定义好的函数直接引用其名字的字符串，或者使用 TensorFlow 和 Theano 自带的激活函数。如果这是整个网络的第一层，则需要用 input_shape 指定输入向量的维度。

(3) 放弃层。放弃层（Dropout）是对该层的输入向量应用放弃策略。在模型训练更新参数的步骤中，网络的某些隐含层节点按照一定比例随机设置为不更新状态，但是权重仍然保留，从而防止过度拟合。这个比例通过参数 rate 设定为 0 到 1 之间的实数。在模型训练时不更新这些节点的参数，因此这些节点并不属于当时的网络；但是保留其权重，因此在以后的迭代次序中可能会影响网络，在打分的过程中也会产生影响，所以这个放弃策略通过不同的参数估计值已经相对固化在模型中了。

(4) 扁平化层。扁化层（Flatten）是将一个维度大于或等于 3 的高维矩阵按照设定“压扁”为一个二维的低维矩阵。其压缩方法是保留第一个维度的大小，然后将所有剩下的数据压缩到第二个维度中，因此第二个维度的大小是原矩阵第二个维度之后所有维度大小的乘积。这里第一个维度通常是每次迭代所需的小批量样本数量，而压缩后的第二个维度就是表达原图像所需的向量长度。

比如输入矩阵的维度为 $(1000, 64, 32, 32)$，扁平化之后的维度为 $(1000, 65536)$，其中 $65536 = 64 \times 32 \times 32$。如果输入矩阵的维度为 $(\text{None}, 64, 32, 32)$，则扁平化之后的维度为 $(\text{None}, 65536)$。

(5) 重构层。重构层（Reshape）的功能和 Numpy 的 Reshape 方法一样，将一定维度的多维矩阵重新排列构造为一个新的保持同样元素数量但是不同维度尺寸的矩阵。其参数为一个元组（tuple），指定输出向量的维度尺寸，最终的向量输出维度的第一个维度的尺寸是数据批量的大小，从第二个维度开始指定输出向量的维度大小。

比如可以把一个有 16 个元素的输入向量重构为一个 $(\text{None}, 4, 4)$ 的新二维矩阵：

```
model = Sequential()
model.add(Reshape( (4, 4), input_shape=(16, ) ) )
```

最后的输出向量不是（4, 4），而是（None, 4, 4）。

(6) 排列层。排列层（Permute）按照给定的模式来排列输入向量的维度。这个方法在连接卷积网络和时间递归网络的时候非常有用。其参数是输入矩阵的维度编号在输出矩阵中的位置。比如：

```
model.add(Permute((1, 3, 2), input_shape=(10, 16, 8)))
```

将输入向量的第二维和第三维的数据进行交换后输出，但是第一维的数据还是待在第一维。这个例子使用了 input_shape 参数，它一般在第一层网络中使用，在接下来的网络层中，Keras 能自己分辨输入矩阵的维度大小。

(7) 向量反复层。顾名思义，向量反复层就是将输入矩阵重复多次。比如下面这个例子：

```
model.add(Dense(64, input_dim=(784, )))
model.add(RepeatVector(3))
```

在第一句中，全连接层的输入矩阵是一个有 784 个元素的向量，输出向量是一个维度为 (one, 64) 的矩阵；而第二句将该矩阵反复 3 次，从而变成维度为 (None, 3, 64) 的多维矩阵，反复的次数构成第二个维度，第一个维度永远是数据批量的大小。

(8) Lambda 层。Lambda 层可以将任意表达式包装成一个网络层对象。参数就是表达式，一般是一个函数，可以是一个自定义函数，也可以是任意已有的函数。如果使用 Theano 和自定义函数，可能还需要定义输出矩阵的维度。如果后台使用 CNTK 或 TensorFlow，可以自动探测输出矩阵的维度。比如：

```
model.add(Lambda(lambda x: numpy.sin(x)))
```

使用了一个现成函数来包装。这是一个比较简单的例子。在 Keras 手册中举了一个更复杂的例子，在这个例子中用户自定义了一个激活函数叫作 AntiRectifier，同时输出矩阵的维度也需要明确定义。

```
def antirectifier(x):
    x -= K.mean(x, axis=1, keepdims=True)
    x = K.l2_normalize(x, axis=1)
    pos = K.relu(x)
    neg = K.relu(-x)
    return K.concatenate([pos, neg], axis=1)

def antirectifier_output_shape(input_shape):
    shape = list(input_shape)
    assert len(shape) == 2
    shape[-1] *= 2
    return tuple(shape)

model.add(Lambda(antirectifier,  output_shape=antirectifier_output_shape))
```

(9) 激活值正则化层。这个网络层的作用是对输入的损失函数更新正则化。

(10) 掩盖层。该网络层主要使用在跟时间有关的模型中，比如 LSTM。其作用是输入张量的时间步，在给定位置使用指定的数值进行“屏蔽”，用以定位需要跳过的时间步。

输入张量的时间步一般是输入张量的第 1 维度（维度从 0 开始算，见例子），如果输入张量在该时间步上等于指定数值，则该时间步对应的数据将在模型接下来的所有支持屏蔽的网络层被跳过，即被屏蔽。如果模型接下来的一些层不支持屏蔽，却接收到屏蔽过的数据，则抛出异常。

```
model = Sequential()
model.add(Masking(mask_value=0., input_shape=(timesteps, features)))
model.add(LSTM(32))
```

如果输入张量 X[batch, timestep, data] 对应于 timestep=5, 7 的数值是 0，即 X[:, [5,7], :] = 0，那么上面的代码指定需要屏蔽的对象是所有数据为 0 的时间步，然后接下来的长短记忆网络在遇到时间步为 5 和 7 的 0 值数据时都会将其忽略掉。

卷积层

针对常见的卷积操作，Keras 提供了相应的卷积层 API，包括一维、二维和三维的卷积操作、切割操作、补零操作等。

卷积在数学上被定义为作用于两个函数 f 和 g 上的操作来生成一个新的函数 z。这个新的函数是原有两个函数的其中一个（比如 f）在另一个（比如 g）的值域上的积分或者加权平均。这可以通过 https://en.wikipedia.org/wiki/Convolution 这个维基百科上的图例来理解。

假设有两个函数 f 和 g，其函数形式如图4.2所示。对这两个函数进行卷积操作按如下步骤进行。

(1) 将 f 和 g 函数都表示为一个变量 t 的函数，如图4.2所示。

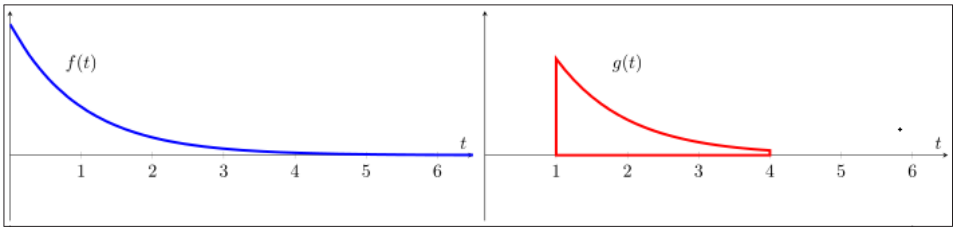


图 4.2 f 和 g 函数形式

(2) 将 f 和 g 函数都表示为虚拟变量 τ 的函数，并将其中一个函数比如 g 的取值进行反转，如图4.3所示。

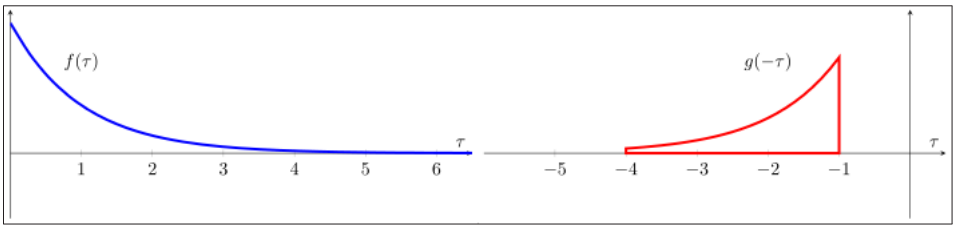


图 4.3　反转 g 的取值

(3) 接下来在此基础上加一个时间抵消项，这样在新的值域上的函数 g 就是在 τ 这个轴上移动的窗口，如图4.4所示的一样。虽然这里是静态图像，但是读者可以想象 g 函数的曲线表示的是该函数沿着坐标轴移动的情景。

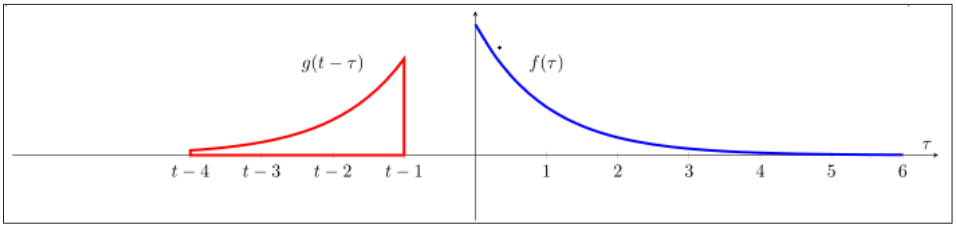


图 4.4　移动 g 函数

(4) 从负无穷大的时间开始，一直移动到正无穷大。在两个函数取值有交接的地方，找出其积分，换句话说，就是对函数 f 计算其在一个平滑移动窗口的加权平均值，而这个权重就是反转后的函数 g 在同样值域中的相应取值。图4.5展示了这个过程。

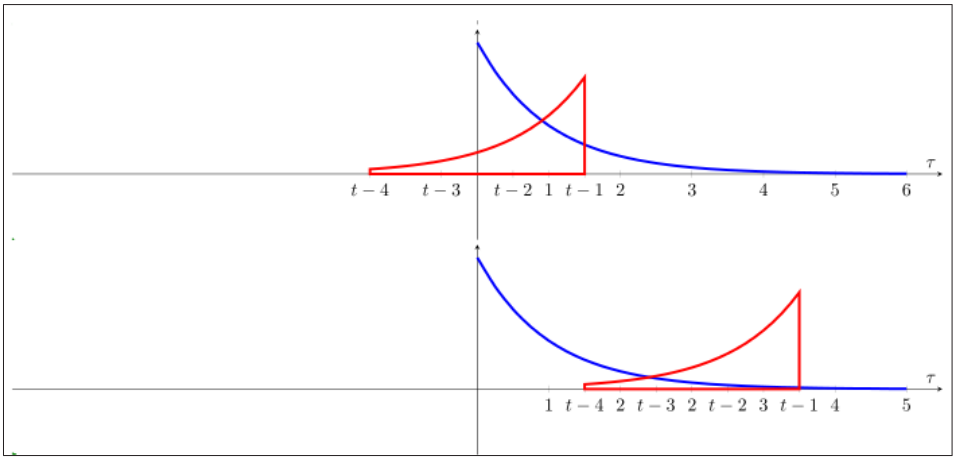


图 4.5　卷积过程

这样得到的波形就是这两个函数的卷积。

卷积操作分为一维、二维和三维，对应的方法分别是 Conv1D、Conv2D 和 Conv3D，这些方法有同样的选项，只是作用于不同维度的数据上，因此适用于不同的业务情景。当作为首层使用时，需要提供输入数据维度的选项 input_shape。这个选项指定输入层数据应有的维度，但是每个维度数据的含义不同，需要分别介绍。

一维卷积通常被称为时域卷积，因为其主要应用在以时间排列的序列数据上，其使用卷积核对一维数据的邻近信号进行卷积操作来生成一个张量。二维卷积通常被称为空域卷积，一般应用在与图像相关的输入数据上，也是使用卷积核对输入数据进行卷积操作的。三维卷积也执行同样的操作。

Conv1D、Conv2D 和 Conv3D 的选项几乎相同。

- filters：卷积滤子输出的维度，要求整数。
- kernel_size：卷积核的空域或时域窗长度。要求是整数或整数的列表，或者是元组。如果是单一整数，则应用于所有适用的维度。
- strides：卷积在宽或者高维度的步长。要求是整数或整数的列表，或者是元组。如果是单一整数，则应用于所有适用的维度。如果设定步长不为 1，则 dilation_rate 选项的取值必须为 1。
- padding：补齐策略，取值为 valid、same 或 causal。causal 将产生因果（膨胀的）卷积，即 output[t] 不依赖于 input[t+1:]，在不能违反时间顺序的时序信号建模时有用。请参考 *WaveNet: A Generative Model for Raw Audio, section 2.1.*。valid 代表只进行有效的卷积，即对边界数据不处理。same 代表保留边界处的卷积结果，通常会导致输出 shape 与输入 shape 相同。
- data_format：数据格式，取值为 channels_last 或者 channels_first。这个选项决定了数据维度次序，其中 channels_last 对应的数据维度次序是（批量数，高，宽，频道数），而 channels_first 对应的数据维度次序为（批量数，频道数，高，宽）。
- activation：激活函数，为预定义或者自定义的激活函数名，请参考前面的“网络层对象”部分的介绍。如果不指定该选项，将不会使用任何激活函数（即使用线性激活函数：$a(x) = x$）。
- dilation_rate：该选项指定扩张卷积（Dilated Convolution）中的扩张比例。要求为整数或由单个整数构成的列表/元组，如果 dilation_rate 不为 1，则步长一项必须设为 1。
- use_bias：指定是否使用偏置项，取值为 True 或者 False。
- kernel_initializer：权重初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的函数。请参考前面的“网络层对象”部分的介绍。

- bias_initializer：偏置初始化方法，为预定义初始化方法名的字符串，或用于初始化偏置的函数。请参考前面的“网络层对象”部分的介绍。
- kernel_regularizer：施加在权重上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- bias_regularizer：施加在偏置项上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- activity_regularizer：施加在输出上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- kernel_constraints：施加在权重上的约束项，请参考前面的关于网络层对象中约束项的介绍。
- bias_constraints：施加在偏置项上的约束项，请参考前面的关于网络层对象中约束项的介绍。

除上面介绍的卷积层以外，还有一些特殊的卷积层，比如 SeparableConv2D、Conv2DTranspose、UpSampling1D、UpSampling2D、UpSampling3D、ZeroPadding1D、ZeroPadding2D、ZeroPadding3D 等，这里限于篇幅就不一一介绍了，感兴趣的读者请参阅 Keras 用户手册。

池化层

池化（Pooling）是在卷积神经网络中对图像特征的一种处理，通常在卷积操作之后进行。池化的目的是为了计算特征在局部的充分统计量，从而降低总体的特征数量，防止过度拟合和减少计算量。举例说明：假设有一个 128 × 128 的图像，以 8×8 的网格做卷积，那么一个卷积操作一共可以得到 $(128-8+1)^2$ 个维度的输出向量，如果有 70 个不同的特征进行卷积操作，那么总体的特征数量可以达到 $70\times(128-8+1)^2=1\,024\,870$ 个。用 100 万个特征做机器学习，除非数据量极大，否则很容易发生过度拟合。所以池化技术就是对卷积出来的特征分块（比如分成新的 $m\times n$ 个较大区块）求充分统计量，比如本块内所有特征的平均值或者最大值等，然后用得到的充分统计量作为新的特征。当然，这个操作依赖于一个假设，就是卷积之后的新特征在局部是平稳的，即在相邻空间内的充分统计量相差不大。对于大多数应用，特别是与图像相关的应用，这个假设可以认为是成立的。图4.6展示了对卷积出来的特征在 4 个（2×2）不重合区块进行池化操作的结果。

Keras 的池化层按照计算的统计量分为最大统计量池化和平均统计量池化；按照维度分为一维、二维和三维池化层；按照统计量计算区域分为局部池化和全局池化。

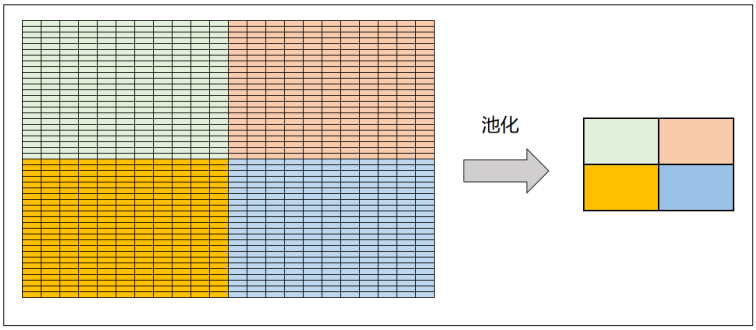


图 4.6 池化操作

(1) 最大统计量池化方法：

- MaxPooling1D，这是对一维的时域数据计算最大统计量的池化函数，输入数据的格式要求为（批量数，时间步，各个维度的特征值），输出数据为三维张量（批量数，下采样后的时间步数，各个维度的特征值）。
- MaxPooling2D，这是对二维的图像数据计算最大统计量的池化函数，输入输出数据均为四维张量，具体的格式根据 data_format 选项要求分别为：

 data_format="channels_first"：输入数据 =（样本数，频道数，行，列），输出数据 =（样本数，频道数，池化后行数，池化后列数）。

 data_format="channels_last"：输入数据 =（样本数，行，列，频道数），输出数据 =（样本数，池化后行数，池化后列数，频道数）。
- MaxPooling3D，这是对三维的时空数据计算最大统计量的池化函数，输入输出数据都是五维张量，具体的格式根据 data_format 选项要求分别为：

 data_format="channels_first"：输入数据 =（样本数，频道数，一维长度，二维长度，三维长度），输出数据 =（样本数，频道数，池化后一维长度，池化后二维长度，池化后三维长度）。

 data_format="channels_last"：输入数据 =（样本数，行，一维长度，二维长度，三维长度），输出数据 =（样本数，池化后一维长度，池化后二维长度，池化后三维长度，频道数）。

(2) 平均统计量池化方法：这个方法的选项和数据格式要求跟最大化统计量池化方法一样，只是池化方法使用局部平均值而不是局部最大值作为充分统计量，方法名字分别为 AveragePooling1D、AveragePooling2D 和 AveragePooling3D。

(3) 全局池化方法：该方法应用全部特征维度的统计量来代表特征，因此会压缩数据维度。比如在局部池化方法中，输出维度和输入维度是一样的，只是特征的维度尺寸

因为池化变小；但是在全局池化方法中，输出维度小于输入维度，如在二维全局池化方法中输入维度为（样本数，频道数，行，列），全局池化以后行和列的维度都被压缩到全局统计量中，因此输出维度只有（样本数，频道数）二维。全局池化方法也分为最大统计量池化和平均统计量池化，以及一维和二维池化方法。

- 一维池化：一维池化方法分为最大统计量和平均统计量两种，方法名字分别为 GlobalMaxPooling1D 和 GlobalAveragePooling1D。输入数据格式要求为（批量数，步进数，特征值），输出数据格式为（批量数，频道数）。这两个方法都没有选项。
- 二维池化：二维池化方法也分为最大统计量和平均统计量两种，方法名字分别为 GlobalMaxPooling2D 和 GlobalAveragePooling2D。这两个方法有关于输入数据要求的选项：data_format。当 data_format="channels_first" 时，输入数据格式为（批量数，行，列，频道数）；当 data_format="channels_last" 时，输入数据格式为（批量数，频道数，行，列）。输出数据格式都为（批量数，频道数）。

循环层

循环层（Recurrent Layer）用来构造跟序列有关的神经网络。但是其本身是一个抽象类，无法实例化对象，在使用时应该使用 LSTM，GRU 和 SimpleRNN 三个子类来构造网络层。在介绍这些子类的用法之前，我们先来了解循环层的概念，这样在写 Keras 代码时方便在头脑中进行映射。循环网络和全连接网络最大的不同是以前的隐藏层状态信息要进入当前的网络输入中。

比如，全连接网络的信息流是这样的：（当前输入数据）⟶ 隐藏层 ⟶ 输出。

而循环网络的信息流是这样的：（当前输入数据 + 以前的隐藏层状态信息）⟶ 当前隐藏层 ⟶ 输出。

下面的例子借用了 iamtrask.github.io 博主的讲解。

图4.7展示了一个典型的循环层依时间步变化的结构。

首先，在时间步为 0 的时候，所有影响都来自于输入，但是从时间步 1 开始，其隐藏层的信息是时间步 0 和时间步 1 的一个混合，时间步 3 的隐藏层状态信息是以前两个时间步和当前时间步信息的混合，依此类推。以前时间步的隐藏层状态信息构成了记忆，因此，网络的大小决定了记忆力的大小，而通过控制哪些记忆来保留和去除可以选择以前时间步的信息对当前时间步的影响力，即记忆的深度。

用上面的信息流方式来表达这个网络，如图4.8所示（此图的彩色效果，读者可以到本书的下载资源中去查看）。

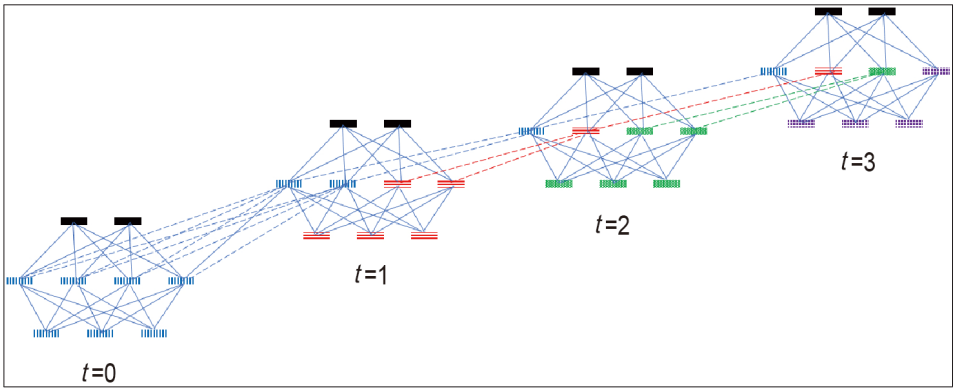


图 4.7 典型的循环层依时间步变化的结构

(input + empty_hidden) -> hidden -> output
(input + prev_hidden) -> hidden -> output
(input + prev_hidden) -> hidden -> output
(input + prev_hidden) -> hidden -> output

图 4.8 循环层结果依时间步变化的信息流表达形式

这里使用色彩形象地显示了在不同时间段的信息通过隐藏层在以后时间中传播和施加影响的过程。

- 简单循环层。SimpleRNN 是循环层的一个子类，用来构造全连接的循环层，是循环网络最直接的应用，使用 recurrent.SimipleRNN 来调用。
- 长短记忆层。LSTM 是循环层的另一个子类，和简单循环层相比，其隐藏状态的权重网络稀疏。
- 带记忆门的循环层（GRU）。

以上具体类别包含如下共同选项。

- units：输出向量的大小，为整数。
- activation：激活函数，为预定义或者自定义的激活函数名，请参考前面的“网络层对象”部分的介绍。如果不指定该选项，将不会使用任何激活函数（即使用线性激活函数：$a(x) = x$）。
- use_bias：指定是否使用偏置项，取值为 True 或者 False。
- kernel_initializer：权重初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的函数。请参考前面的“网络层对象”部分的介绍。

- recurrent_initializer：循环层状态节点权重初始化方法，为预定义初始化方法名的字符串，或用于初始化权重的函数。请参考前面的“网络层对象”部分的介绍。
- bias_initializer：偏置初始化方法，为预定义初始化方法名的字符串，或用于初始化偏置的函数。请参考前面的“网络层对象”部分的介绍。
- kernel_regularizer：施加在权重上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- recurrent_regularizer：施加在循环层状态节点权重上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- bias_regularizer：施加在偏置项上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- activity_regularizer：施加在输出上的正则项，请参考前面的关于网络层对象中正则项的介绍。
- kernel_constraint：施加在权重上的约束项，请参考前面的关于网络层对象中约束项的介绍。
- recurrent_constraint：施加在循环层状态节点权重上的约束项，请参考前面的关于网络层对象中约束项的介绍。
- bias_constraint：施加在偏置项上的约束项，请参考前面的关于网络层对象中约束项的介绍。
- dropout：指定输入节点的放弃率，为 0 到 1 之间的实数。
- recurrent_dropout：指定循环层状态节点的放弃率，为 0 到 1 之间的实数。

LSTM 和 GRU 则额外包含一个选项叫作 recurrent_activation，这个选项控制循环步所使用的激活函数。

嵌入层

嵌入层（Embedding Layer）是使用在模型第一层的一个网络层，其目的是将所有索引标号映射到致密的低维向量中，比如 [[4], [32], [67]] → [[0.3, 0.9, 0.2], [-0.2, 0.1, 0.8], [0.1, 0.3, 0.9]] 就是将一组索引标号映射到一个三维的致密向量中，通常用在对文本数据进行建模的时候。输入数据要求是一个二维张量：（批量数，序列长度），输出数据为一个三维张量：（批量数，序列长度，致密向量的维度）。

其选项如下。

- 输入维度：这是词典的大小，一般是最大标号数 +1，必须是正整数。

- output_dim：输出维度，这是需要映射到致密的低维向量中的维度，为大于或等于 0 的整数。
- embeddings_initializer：嵌入矩阵的初始化方法，请参考前面的关于网络层对象中对初始化方法的介绍。
- embeddings_regularizer:：嵌入矩阵的正则化方法，请参考前面的关于网络层对象中正则项的介绍。
- embeddings_constraint：嵌入层的约束方法，请参考前面的关于网络层对象中约束项的介绍。
- mask_zero：是否屏蔽 0 值。通常输入值里的 0 是通过补齐策略对不同长度输入补齐的结果，如果为 0，则需要将其屏蔽。如果输入张量在该时间步上都等于 0，则该时间步对应的数据将在模型接下来的所有支持屏蔽的网络层被跳过，即被屏蔽。如果模型接下来的一些层不支持屏蔽，却接收到屏蔽过的数据，则抛出异常。如果设定了屏蔽 0 值，则词典不能从 0 开始做索引标号，因为这时候 0 值已经具有特殊含义了。
- input_length：输入序列长度。当需要连接扁平化和全连接层时，需要指定该选项；否则无法计算全连接层输出的维度。

合并层

合并层是指将多个网络产生的张量通过一定方法合并在一起，可以参看下一节中的奇异值分解的例子。合并层支持不同的合并方法，包括：元素相加（merge.Add）、元素相乘（merge.Multiply）、元素取平均（merge.Average）、元素取最大（merge.Maximum）、叠加（merge.Concatenate）、矩阵相乘（merge.Dot）。

其中，元素相加、元素相乘、元素取平均、元素取最大方法要求进行合并的张量的维度大小完全一致。叠加方法要求指定按照哪个维度（axis）进行叠加，除了叠加的维度，其他维度的大小必须一致。矩阵相乘方法是对两个张量采用矩阵乘法的形式来合并，因为张量是高维矩阵，因此需要指定沿着哪个维度（axis）进行乘法操作。同时可以指定是否标准化（Normalize），如果是的话（Normalize=True），则先将两个张量归一化以后再相乘，这时得到的是余弦相似度。

来自于 *MIT Technology Review* 的图4.9很好地展示了网络合并结构。

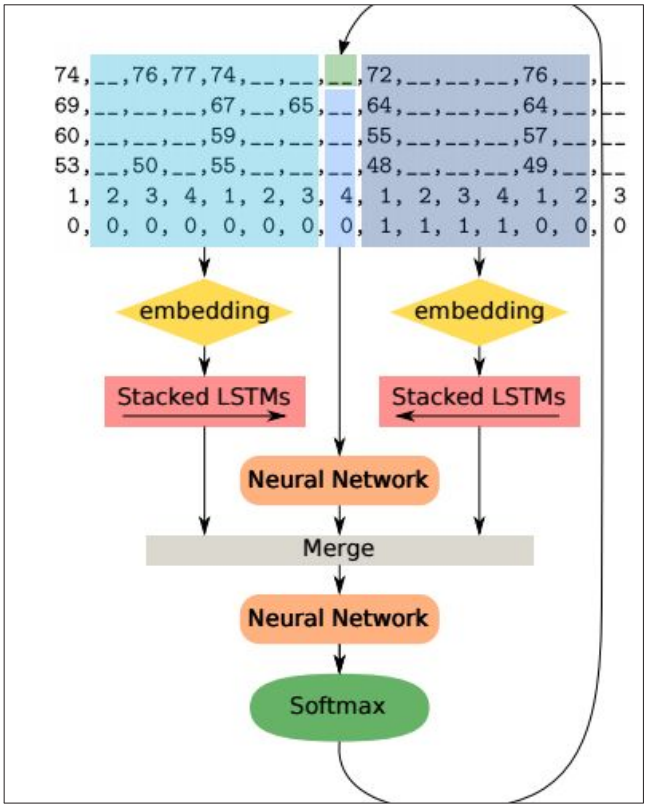


图 4.9　网络合并结构

4.6　使用 Keras 进行奇异值矩阵分解

Keras 虽然是针对深度学习的各种模型设计的，但是通过巧妙构造网络结构，可以实现特定的传统算法。下面介绍使用 Keras 来进行奇异值矩阵分解（SVD）。

奇异值矩阵分解是一种基本的数学工具，被应用于大量的数据挖掘算法中，比较有名的有协同过滤（Collaborative Filtering），PCA 回归等算法。矩阵分解的目的是解析矩阵的结构，提取重要信息，去除噪声，实现数据压缩等。比如在奇异值矩阵分解中，信息都集中在头几个特征向量中，使用这几个向量有可能较好地（即均方差尽可能小地）复原原来的矩阵，同时只需要保留较少的数据。

这里我们介绍使用 Keras 进行奇异值矩阵分解技巧。比如 SVD 被应用于协同过滤推荐算法中，主要目的是用来对数据降维，提高计算速度。协同过滤算法一般应用于用户–物品矩阵。图4.10展示了一个简单的这种矩阵。在这个矩阵中每一行代表一个用户，

每一列代表一个物品，因此在每一行中标注了用户历史上对该物品的评价或者购买情况。如果没有购买过或者没有评价该物品，则数值为空值，有时候也用 0 代替；如果购买过该物品，那么数值一般是 1 或者购买次数；如果是评分，则通常为实际评分，比如为 1~5 分。

	物品1	物品2	物品3	物品4	物品5	物品6	物品7	…
用户1		1				5		
用户2			2					
用户3					3			
用户4		2			3			3
用户5						2	1	
用户6				4				
用户7		2	1		5	3		
用户8							3	5
用户9				1		2		
用户10				1	1		1	
用户11						4	2	
用户12			4		4			3
用户13								1
用户14						5	1	
用户15			5		3			
用户16		1						
用户17			3					5
用户18					2	2		
用户19		1	4	3				
用户20				1		5		1
…								

图 4.10　用户-物品矩阵图

SVD 基于以下线性代数定理：任何 $m \times n$ 的实数矩阵 $\boldsymbol{X}$ 可以表示为如下三个矩阵的乘积：$m \times r$ 的酉矩阵 $\boldsymbol{U}$，被称为左特征向量矩阵；$r \times r$ 的对角阵 $\boldsymbol{S}$，被称为特征值矩阵；$r \times n$ 的酉矩阵 $\boldsymbol{V}^{\mathrm{T}}$，被称为右特征向量矩阵，其中 $r \leqslant n$。

$$\boldsymbol{X}_{m\times n} = \boldsymbol{U}_{m\times r}\boldsymbol{S}_{r\times r}\boldsymbol{V}^{\mathrm{T}}_{r\times r}$$

上述矩阵分解可以用图4.11直观地表达。

$$\underset{m\times n}{\boldsymbol{X}} = \underset{m\times r}{\boldsymbol{U}} \quad \underset{r\times r}{\boldsymbol{S}} \quad \underset{r\times n}{\boldsymbol{V}^{\mathrm{T}}}$$

$$\begin{pmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & \\ \vdots & \vdots & \ddots & \\ x_{m1} & & & x_{mn} \end{pmatrix} = \begin{pmatrix} u_{11} & \cdots & u_{1r} \\ \vdots & \ddots & \\ u_{m1} & & u_{mr} \end{pmatrix} \begin{pmatrix} s_{11} & 0 & \cdots \\ 0 & \ddots & \\ \vdots & & s_{rr} \end{pmatrix} \begin{pmatrix} v_{11} & \cdots & v_{1n} \\ \vdots & \ddots & \\ v_{r1} & & v_{rn} \end{pmatrix}$$

图 4.11　奇异值分解演示

我们看到，其实 SVD 是将原始矩阵分解为一个对应于行信息的矩阵 $\boldsymbol{U}$ 和对应于列信息的矩阵 $\boldsymbol{V}$，因为包含特征值的对角阵可以取方根后分别纳入左、右特征向量矩阵中，所以现在要将原始矩阵分解为两个致密的实数矩阵，使得它们的乘积和原始矩阵的均方差尽量小。我们在使用 Keras 来操作矩阵分解时也是遵循这个思路的，使用的工具就是 Keras 层里面的嵌入（Embedding）工具和合并（Merge）工具。嵌入工具能够将

一组正整数（比如序列的索引）转换为固定维度的致密实数，而合并工具能够按照不同的方法，比如求和、叠加或者乘积方式将两个网络合并在一起。

- 首先，我们将用户和物品各自编号，就能使用嵌入工具将用户和物品各自映射到一个固定的空间中。比如在下面的示例代码中，第 1 行和第 2 行先定义用户序列的输入，然后使用嵌入工具将 n_users 个用户里的每一人投影到 n_factor 维的新空间中。一开始在新空间中的位置是随机的，即初始化是一个随机向量，以后在模型拟合阶段再求得最优解。
- 其次，将各自在新空间中的投影使用乘积方式合并起来，就能得到拟合后的 UV' 乘积矩阵：x = merge([u, v], mode='dot')，其中 mode='dot' 表示使用矩阵乘法来合并两个矩阵。
- 最后，定义一个模型，并使用随机梯度递减算法来拟合，使得这个乘积矩阵和原始矩阵的均方差（MSE）最小。可以通过下面的命令来实现：

```
model = Model([user_in, movie_in], x);
model.compile(Adam(0.001), loss='mse')
```

下面是完整的程序。

```
user_in = Input(shape=(1,), dtype='int64', name='user_in')
u = Embedding(n_users, n_factors, input_length=1)(user_in)
movie_in = Input(shape=(1,), dtype='int64', name='movie_in')
v = Embedding(n_movies, n_factors, input_length=1)(movie_in)

x = merge([u, v], mode='dot')
x = Flatten()(x)
model = Model([user_in, movie_in], x)
model.compile(Adam(0.001), loss='mse')

model.fit([trn.userId, trn.movieId], trn.rating, batch_size=64, epochs=1))
```

5 推荐系统

5.1 推荐系统简介

推荐系统是机器学习最广泛的应用领域之一，大家熟悉的亚马逊、迪士尼、谷歌、Netflix 等公司都在网页上有其推荐系统的界面，帮助用户更快、更方便地从海量信息中找到有价值的信息。比如亚马逊（www.amazon.com）会给你推荐书、音乐等，迪士尼（video.disney.com）给你推荐最喜欢的卡通人物和迪士尼电影，谷歌搜索更不用说了，Google Play、Youtube 等也有自己的推荐引擎、推荐视频和应用等。

亚马逊和谷歌的推荐网页截图分别如图5.1和图5.2所示。



图 5.1　亚马逊推荐网页

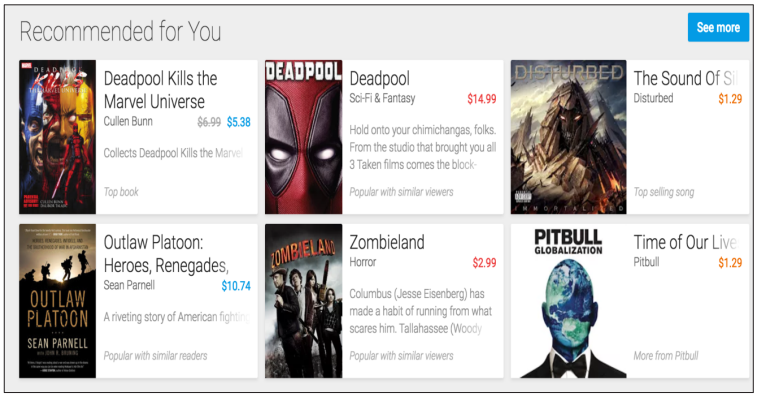


图 5.2　谷歌应用商城推荐网页

推荐系统的最终目的是从百万甚至上亿内容或者商品中把有用的东西高效地显示给用户，这样可以为用户节省很多自行查询的时间，也可以提示用户可能忽略的内容或商品，使用户更有黏性，更愿意花时间待在网站上，从而使商家可以从内容或者商品中赚取更多的利润，即使流量本身也会使商家从广告中受益。那么推荐系统背后的魔术是什么呢？其实我们可以这么想，任何推荐系统本质上都是在做排序的问题。把系统里所有的音乐、电影、应用等从高到低进行喜好排序，把排名高的推荐给用户，用户喜欢了，推荐系统自然就会有价值。读者可能注意到了，排序的前提是对喜好的预测。那么喜好的数据从哪里来呢？这里有几个渠道，比如你和产品有过互动，看过亚马逊商城的一些书，或者买过一些书，那么你的偏好就会被系统学到，系统会基于一些假设给你建立画像和构建模型。你和产品的互动越多，数据点就越多，画像就越全面。除此之外，如果你有跨平台的行为，那么各个平台的数据汇总，也可以综合学到你的偏好。比如谷歌搜索、地图和应用商城等都有你和谷歌产品的互动信息，这些平台的数据可以通用，应用的场景有很大的想象力。平台还可以利用第三方数据，比如订阅一些手机运营商的数据，用来多维度刻画用户。总之，在现代互联网科技时代，数据是最根本的。

这里会有几个采集和处理数据的关键点。

(1) 首先，需要理解用户的数据。比如用户点过了某个视频，那到底算喜欢还是不喜欢呢？一种解决方法是合理定义训练数据的喜好分值，比如用户不仅点了，还停留了不少时间，或者视频播放时中间没有跳放，或者用户看了好几遍，或者用户点赞了，这些都是正面信号。负面信号与之类似，也可以定义。这里就需要用统计的方法定义一些好的标注和差的标注。正面的标注就会在后续模型中拿去最大化，负面的指标则需要最小化。

(2) 其次，需要合理看待和处理缺失数据。比如平台做得不够完善，数据丢了，或者有些数据如年龄、地区等可填可不填，如果大部分用户不填，那么平台就缺失了很多这类数据。而且更令人烦躁的是，这些缺失很多时候是带有偏见的，不是随机缺失，这给后续分析造成了一定的困难。当然，有些数据即使缺失也是可以预测的，比如看动画片，那么可以推测该用户是小孩或者年轻人的可能性比较大。这里需要更深层次的建模来还原一些数据。

(3) 再次，需要打通各平台之间数据的联系。平台一般会给每个用户一个唯一的ID，可能基于账号、设备或者浏览器的 Cookie。不同的平台之间用户 ID 可能不一样，如果利用第三方数据的话，ID 几乎不可能对得上。这里就面临一个数据整合和打通的问题。

一般有两种处理方法。一种是利用其他信息，比如 IP 地址、设备类型等特征近似地把两方面的数据匹配起来。更复杂一点的是，利用模型去预测两个平台的用户如何配对。另一种是除去 ID 的信息，即匿名。这也很常见，因为用户的浏览行为不一定需要登录，可以换个设备浏览等。我们聚焦在用户行为上，从高度分析用户一系列行为之间的规律，而不去追究具体哪个用户以前干了什么。比如，第一个用户看了 A 之后又看了 B，第二个用户同样看了 A 之后也看了 B，不论这两个用户是不是同一个人，我们都知道 A 和 B 之间存在联系。从这个意义上讲，这也提供了数据价值。再比如，如果第二个用户只看了 A，那么是不是可以认为 B 也有很大的概率被这个用户喜欢呢？互联网公司必须非常注重保护用户数据和隐私。匿名的另一个好处是记录信息越少，对用户来说信息安全保障就越大。

(4) 最后，就模型而言，在大数据环境下可以非常复杂。除了用户和内容商品之间的交互行为，还有如何把年龄、内容标签等加入模型中；用户的兴趣点可能和时间有关，尤其在阅读新闻方面，用户一般更愿意读新出炉的文章，而不是老掉牙的文章等。这里就牵涉到如何把时间因素考虑进去等。

模型只是推荐系统的一部分。我们建完推荐系统模型，还必须考虑如下几个工程上的实践问题。

第一，模型如何实时调用。好的推荐系统需要有实时性，不论在数据还是计算上。如果数据更新太慢，或者模型无法把最新的用户信息包含进去，推荐就大打折扣了。这里就可能需要开发在线更新模型，同时也需要数据以流的形式进入数据库和模型。在计算上，什么数据需要在内存中存储等也需要考虑。

第二，模型调用如何保证低延迟。用户不可能为了推荐系统结果等半天，系统需要快速反应。这里推荐系统平台需要实施大规模调度、平衡负载和压力测试。

第三，模型评判标准怎么设。现在通常是用信息获取中的精确率（Precision）和召回率（Recall）来作为指标。但是也有一些更丰富的指标，比如平均百分位数（Mean Percentile Rank）等。

第四，新建的模型怎么上线。一般大家都会做实验，先给百分之一的人用新建的推荐系统，用另外百分之一的人做对照组。在统计意义上，如果新建的模型效果好，就可以慢慢把新建的模型推广到百分之五、百分之十等，最终推广到百分之百，即所谓的完全上线。平台拿到的用户日志数据时间不一，使用的指标通常必须是短期的，而不是最终需要提升的长期指标（比如留存率等），因为实验根本等不到长期指标出来的那一刻。怎么设定指标也是个学问。指标因为样本而变化，怎么去除噪声、怎么处理稀疏数据、怎么描述统计显著性等是实验设计的重中之重。还有，设定的短期指标必须和长期指标方向一致，怎么找好的短期指标也需要花时间去探索和通过数据去验证。

第五，还需要设计监控系统。万一指标出现异常，这是就需要有一套机制进行异常检测，查找原因，并可以迅速返回到前一个版本等。监控系统的搭建就需要结合异常检测等机器学习模型，在此不再一一叙述。

本章节重点讲两个推荐系统的重要算法：矩阵分解模型和深度模型。然后，我们会讨论一些常用的指标用来评价模型的好坏。

5.2 矩阵分解模型

矩阵分解其实是数学上的一个经典问题。大家从线性代数中可以知道，矩阵可以做 SVD 分解、Cholesky 分解等，就好比任何大于 1 的正整数都可以分解成若干质数的乘积。这里讲的矩阵分解是指，对于任何一个矩阵 $\boldsymbol{P}_{m\times n}$，是否可以找到低维度的两个矩阵 $\boldsymbol{A}_{m\times k}$ 和 $\boldsymbol{B}_{k\times n}$ 的乘积去近似。我们希望 k 比较小，因为如果 k 很大就没什么意义了，比如 $\boldsymbol{A}$ 直接可以等于 $\boldsymbol{P}$，$\boldsymbol{B}$ 等于单位矩阵，那么 $\boldsymbol{A}\times\boldsymbol{B}=\boldsymbol{P}$。这样信息虽然不会丢失，但没有给我们丝毫帮助。

矩阵分解可以认为是一种信息压缩。这里有两种理解。第一种理解，用户和内容不是孤立的，用户喜好有相似性，内容也有相似性。压缩是把用户和内容数量化，压缩成 k 维的向量。那么读者可能会问，采用 One Hot 编码，即把用户表示成 0,1 形式的向量，为什么不好呢？因为 One Hot 编码需要的空间太大，比如有 10 亿用户，就必须用 10 亿维的向量去表示。另外，这些向量之间没有任何联系。反过来说，把用户向量维度进行压缩，使得向量维度变小，本身就是信息压缩的一种形式；向量之间还可以进行各种计算，比如余弦（Cosine）相似性，就可以数量化向量之间的距离、相似度等。第二种理解，从深度学习的角度，用户表示输入层（User Representation）通常用 One Hot

编码，这没问题，但是通过第一层全连接神经网络就可以到达隐藏层，就是所谓的嵌入层（Embedding Layer），也就是我们之前提到的向量压缩过程。紧接着这个隐藏层，再通过一层全连接网络就是最终输入层，通常用来和实际标注数据进行比较，寻找差距，用来更新网络权重。从这个意义上讲，完全可以把整个数据放进神经系统的框架中，通过浅层学习把权重求出来，就是我们要的向量集合了。

经过这么分析，矩阵分解在推荐系统中是如何应用的就显而易见了。假设有用户和内容（比如电影）的互动数据，其中一种情况是 Netflix 的打分模式，即用户会给电影进行 1~5 的打分；另一种情况是基于用户行为的，比如用户是否看了某部电影、看了多长时间等。通常第二种模式更加值得信赖，因为对于打分，一来每个人的评判标准不同；二来很多人即使看了电影也不打分，即使打分，也有可能只是对自己满意或者不满意的电影打分，以至于很容易造成系统性偏差，后期处理起来比较复杂。反过来说，用户看电影的行为被机器日志所记录，是真实的数据，不需要担心数据不准确或者有偏差的问题。

这两种情形都可以用矩阵分解来解决。假设数据库里有 m 个用户和 n 部电影，那么用户电影矩阵的大小就是 $m \times n$。每个单元 (i, j) 用 $\boldsymbol{R}_{ij}$ 表示用户是否看了该电影，即 0 或 1。我们把用户和电影用类似 Word2Vec 的方法分别进行向量表示，把每个用户 i 表示成 d 维向量 X_i，把每部电影 j 表示成 d 维向量 Y_j。我们要寻找 X_i 和 Y_j，使得 $X_i \times Y_j$ 和用户电影矩阵 $\boldsymbol{R}_{ij}$ 尽可能接近，如图5.3所示。这样对于没出现过的用户电影对，通过 $X_i \times Y_j$ 的表达式可以预测任意用户对电影的评分值。

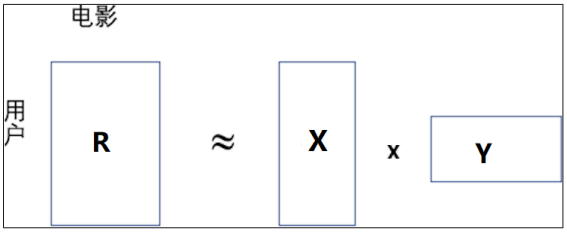


图 5.3 矩阵分解

注意：这里 d 是一个远小于 m, n 的数。从机器学习的角度来说，模型是为了抓住数据的主要特征，去掉噪声。越复杂、越灵活的模型带来的噪声越多，降低维度则可以有效地避免过度拟合现象的出现。

用数学表达式可以这么写：$\sum_{i,j}(r_{ij} - X_i \times Y_j)^2$

一般为了进一步避免过度拟合，还会加入正则项。为了便于优化（求导）计算，通常使用 L_2。但是在实际应用过程中，L_2, L_1 等更复杂的正则项都可以使用。对于正则项的选择，一来是适合计算的需要，加入的表达式要便于优化计算；二来是基于对模型

的假设，比如假设某些系数相等，或者某些系数同为零或同不为零等。但是最终目的都是为了简化模型，避免过度拟合，从而达到更好的普适性。

加入正则项之后的表达式可以写成：

$$\sum_{i,j}(r_{ij} - X_i \times Y_j)^2 + \lambda(\sum_i ||X_i||^2 + \sum_j ||Y_j||^2)$$

其中，λ 是可以调节的参数，用来控制惩罚的程度。如果 λ 很大，那么所有 X_i, Y_j 都得为 0; 反之，如果 λ 很小，那么 X_i, Y_j 的选择余地就比较大，好比没有约束。

搞清楚了原理后，我们就可以开始用 Keras 实现矩阵分解代码了。推荐系统最经典的公开数据就是 Movielens，该数据集有 100 万个评分数据。我们用这些数据来演示如何构建推荐系统模型。

首先加载 Keras 必需的包和搭建神经网络的模块。

```
import math
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Embedding, Dropout, Dense, Merge
```

搭建深度学习神经网络模型的基本思想是这样的：从直观上说，我们是要对任意用户和电影组合进行评分预测。输入是用户和电影对，输出是评分。我们把用户和电影分别用嵌入层的向量来表示，这样实际的输入层就是用户向量和电影向量，实际的输出层就是评分。由于评分范围是 1~5，预测时可以把它当连续变量，直接预测值就可以了。也可以把问题看成分类问题，最后一层用 Softmax 建，损失函数用交叉熵（Cross Entropy）的标准就可以了。

那么按照矩阵分解的思路，深度学习模型怎么搭建呢？可以这样：到了嵌入层，只需要把两个向量乘起来，和已知的评分做个比较，如果有偏差，就用向后传播的方法，调整嵌入层的向量，直到最后预测评分和已知评分比较接近。

当然，在实际的操作过程中，也可以加入 Dropout 等技术防止过度拟合。

首先选择嵌入层的维度，这里为 128。但这是一个可以调节的参数，一般在几百范围内比较合适。Word2Vec 用的是 300 维。

我们用 Pandas 读取数据，并且计算一些数据统计量，比如有多少个用户、多少部电影等。这里用户和电影都是建过索引的，从 1 开始。在实践过程中数据一般没有索引，需要读者自行创建。

```
k = 128
ratings = pd.read_csv("ratings.dat", sep = '::', names = ['user_id','
movie_id','rating','timestamp'])
n_users = np.max(ratings['user_id'])
n_movies = np.max(ratings['movie_id'])
print([n_users, n_movies, len(ratings)])
```

通过简单的分析，我们知道数据集有 6040 个用户、3952 部电影和 1 000 209 个评分。

我们看一下数据集有多稀疏：

$$1000209/(6040.0 * 3852.0) = 4.29\%$$

这说明只有 4.29% 的用户电影组合有评分。矩阵大部分数据都是缺失的。上文提到，这符合常理，毕竟平均每个用户看的电影数量有限，看过的也未必会打分。

评分的分布是怎样的呢？下面的代码展示了评分的分布。

```
plt.hist(ratings['rating'])
plt.show()
print(np.mean(ratings['rating']))
```

所有评分的平均分为 3.58。大多数分数都集中在 3~5 之间，如图5.4所示。

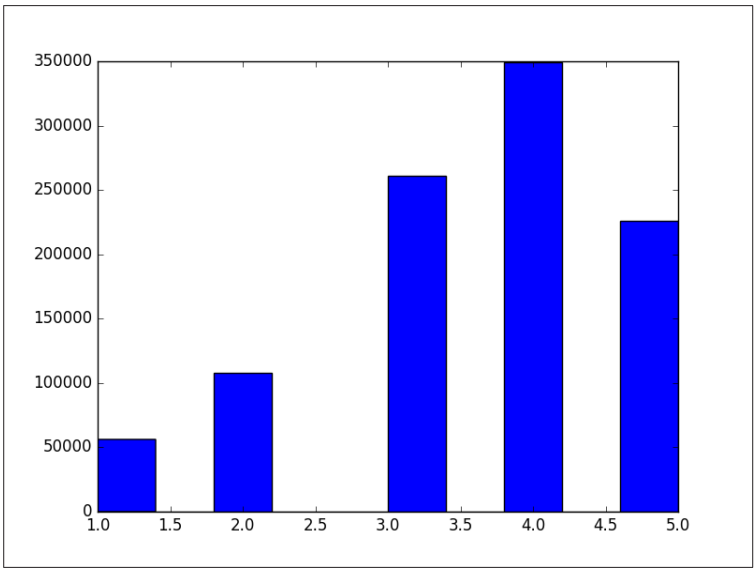


图 5.4 评分分布柱形图

读者还可以做一些其他分析，比如哪些用户系统性地打分偏高等。

接下来可以建模型了。由于这个模型的特殊结构，我们可以建两个小神经网络，然后用第三个小神经网络对前两个小神经网络的输入做运算，如图5.5所示。

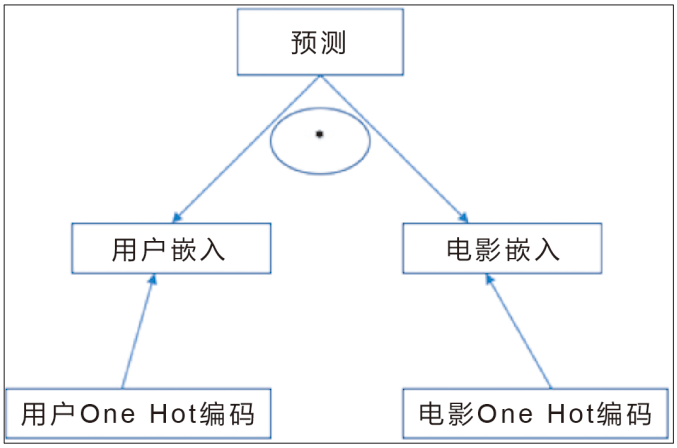


图 5.5　基于矩阵分解的神经网络模型

第一个小神经网络，处理用户嵌入层。注意到，用户嵌入层的第一个参数必须比最大的索引值大，第二个参数指的是嵌入层的维度，第三个参数指的是每次输入数据时会用到嵌入层的几个索引，一般这个数字是固定的。在文本情感分析中，我们也使用了类似的技术。

```
model1 = Sequential()
model1.add(Embedding(n_users + 1, k, input_length = 1))
model1.add(Reshape((k,)))
```

第二个小神经网络，处理电影嵌入层。

```
model2 = Sequential()
model2.add(Embedding(n_movies + 1, k, input_length = 1))
model2.add(Reshape((k,)))
```

第三个小神经网络，在第一、二个网络的基础上叠加乘积运算。

```
model = Sequential()
model.add(Merge([model1, model2], mode = 'dot', dot_axes = 1))
```

输出层和最后的评分数进行对比，后向传播去更新网络参数。

```
model.compile(loss = 'mse', optimizer = 'adam')
```

也可以尝试用 RMSPROP 或 ADAGRAD 算法。关于这两个算法，可以参考 http://cs231n.github.io/neural-networks-3/#update 网站中的介绍。

```
model.compile(loss = 'mse', optimizer = 'rmsprop')
model.compile(loss = 'mse', optimizer = 'adagrad')
```

下面我们获取用户索引数据和电影索引数据，不过相应的特征矩阵 X_train 需要两个索引数据一起构造。

```
users = ratings['user_id'].values
movies = ratings['movie_id'].values
X_train = [users, movies]
```

评分数据可以以如下方式获取。

```
y_train = ratings['rating'].values
```

一切准备就绪以后，我们用大小为 100 的小批量，使用 50 次迭代来更新权重。这两个参数也可以调整。一般批量大小为几百，迭代次数可以达到上百到几百范围。损失一般一开始会下降得比较快，随后慢慢下降。通常做法是等损失稳定下来后再结束训练会比较好。

```
model.fit(X_train, y_train, batch_size = 100, epochs = 50)
```

模型训练完以后，我们可以预测未给的评分。比如采用如下代码，可以预测第 10 个用户对编号 99 的电影的评分。

```
i=10
j=99
pred = model.predict([np.array([users[i]]), np.array([movies[j]])])
```

计算训练样本误差。

```
sum = 0
for i in range(ratings.shape[0]):
    sum += (ratings['rating'][i] - model.predict([np.array([ratings['user_id'
    ][i]]), np.array([ratings['movie_id'][i]])])) ** 2
mse = math.sqrt(sum/ratings.shape[0])
print(mse)
```

结果显示，在训练数据集上，拟合误差比较小，只有 0.34。

这里只是做一个演示，在实际建模中应该把数据按时间轴分成训练数据、校对数据和测试数据，从而正确地评价模型的好坏。训练数据拟合得好，只能说明算法本身是在做正确的优化事情，并不能说明模型在未知的数据集上是否是好的，也不能说明模型抓住了本质，排除了噪声。

关于如何评价模型好坏，以后会提到。我们也可以在上述神经网络中加入 Dropout 等技术，后面将会演示一个进阶版的深度学习网络模型。

上述是神经网络模型在矩阵分解算法上的实施，其实本质上只是借用了神经网络的优化算法和结构，计算出了矩阵分解。关于矩阵分解，有更简便的办法去计算，这里介绍一种交替最小二乘法（ALS）。

类似于 Dropout 的正则方式，在矩阵分解中一般也会对分解出来的矩阵做限制，比如加 L_1, L_2，可以写成这样的形式：$(\boldsymbol{M}-\boldsymbol{A}\times\boldsymbol{B})^2+\lambda(L_2(\boldsymbol{A})+L_2(\boldsymbol{B}))$。

交替最小二乘法的想法很简单，我们要解决的是分解矩阵 $\boldsymbol{M}$ 近似等于两个新矩阵 $\boldsymbol{A}$ 和 $\boldsymbol{B}$ 的乘积，限制条件是 $\boldsymbol{A}$、$\boldsymbol{B}$ 的值不能太大，并且部分 $\boldsymbol{M}$ 的数据已知。类似于坐标下降法（Coordinate Descent）的想法，我们可以先固定 $\boldsymbol{A}$，这样求 $\boldsymbol{B}$ 就是一个最小二乘法的问题。类似的，得到了 $\boldsymbol{B}$ 以后固定 $\boldsymbol{B}$，再求 $\boldsymbol{A}$，循环迭代。如果最后 $\boldsymbol{A}$、$\boldsymbol{B}$ 都收敛，即它们在两次迭代间的变换小于一个阈值时，就可以认为找到了问题的解。

对于学统计的同学，对唯一性的概念比较敏感。比如做线性回归，当自变量之间的相关性很高时，解的唯一性就会有问题。在矩阵分解这个问题上，很遗憾解的唯一性无法克服，因为 $\boldsymbol{A}$ 或者 $\boldsymbol{B}$ 都可以乘上一个正交矩阵 $\boldsymbol{T}$，这样 $\boldsymbol{A}\times\boldsymbol{T}\times\boldsymbol{T}^{\mathrm{T}}\times\boldsymbol{B}$ 也是解。

那么有读者要问，这到底会不会影响解？答案是不会。因为最后做预测用的是分解完矩阵的乘积，无论是 $\boldsymbol{A}\times\boldsymbol{B}$ 还是 $\boldsymbol{A}\times\boldsymbol{T}\times\boldsymbol{T}^{\mathrm{T}}\times\boldsymbol{B}$，都是一样的结果。

5.3 深度神经网络模型

下面展示进阶版的深度模型。我们将建立多层深度学习模型，并且加入 Dropout 技术。

这个模型非常灵活。因为如果有除用户、电影之外的数据，比如用户年龄、地区、电影属性、演员等外在变量，则统统可以加入模型中，用嵌入的思想把它们串在一起，作为输入层，然后在上面搭建各种神经网络模型，最后一层可以用评分等作为输出层，这样的模型可以适用于很多场景。

另外值得一提的是，谷歌有篇非常出色的研究论文，讲的是宽深模型（Wide and Deep Model）。具体论文见：https://arxiv.org/abs/1606.07792。宽深模型适用的场合是有

多个特征，有些特征需要用交叉项特征合成（宽度模型），而有些特征需要进行高维抽象（深度模型）。宽深模型很好地结合了宽度模型和深度模型，同时具有记忆性和普适性，从而提高准确率。

宽深模型的结构如图5.6所示。

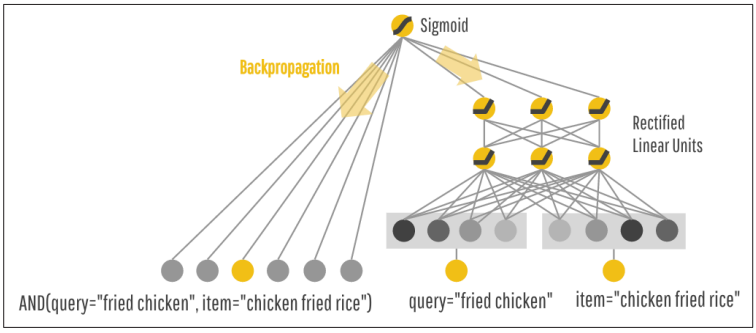


图 5.6　宽深模型结构

宽深模型比本章所要阐述的模型多了一个内容，就是交叉项。我们之所以用深度模型，是因为数据基本只涉及用户、电影和打分，宽深模型无法很好地演示。换句话说，我们的数据集更适合深度模型，而不适合宽度模型。不过，我们可以通过宽深模型的架构和图5.7展示的深度模型对如何用深度学习做推荐有更好的把握。

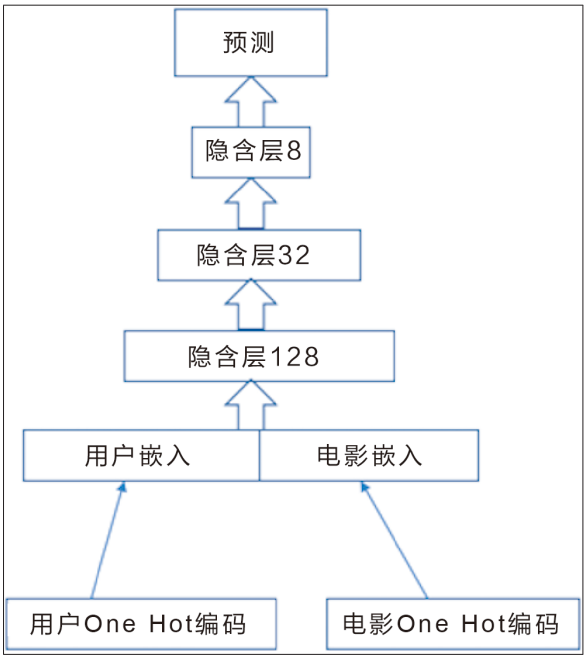


图 5.7　深度模型

首先，做用户和电影的嵌入层。

```
k = 128
model1 = Sequential()
model1.add(Embedding(n_users + 1, k, input_length = 1))
model1.add(Reshape((k,)))
model2 = Sequential()
model2.add(Embedding(n_movies + 1, k, input_length = 1))
model2.add(Reshape((k,)))
```

第三个小神经网络，在第一、二个网络的基础上把用户和电影向量结合在一起。

```
model = Sequential()
model.add(Merge([model1, model2], mode = 'concat'))
```

然后加入 Dropout 和 relu 这个非线性变换项，构造多层深度模型。

```
model.add(Dropout(0.2))
model.add(Dense(k, activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(int(k/4), activation = 'relu'))
model.add(Dropout(0.5))
model.add(Dense(int(k/16), activation = 'relu'))
model.add(Dropout(0.5))
```

因为是预测连续变量评分，最后一层直接上线性变化。当然，读者可以尝试分类问题，用 Softmax 去模拟每个评分类别的概率。

```
model.add(Dense(1, activation = 'linear'))
```

将输出层和最后的评分数进行对比，后向传播去更新网络参数。

```
model.compile(loss = 'mse', optimizer = "adam")
```

接下来要给模型输入训练数据。

首先，收集用户索引数据和电影索引数据。

```
users = ratings['user_id'].values
movies = ratings['movie_id'].values
```

收集评分数据。

```
label = ratings['rating'].values
```

构造训练数据。

```
X_train = [users, movies]
y_train = label
```

然后，用小批量更新权重。

```
model.fit(X_train, y_train, batch_size = 100, epochs = 50)
```

模型训练完以后，预测未给的评分。

```
i,j = 10,99
pred = model.predict([np.array([users[i]]), np.array([movies[j]])])
```

最后，对训练集进行误差评估。

```
sum = 0
for i in range(ratings.shape[0]):
    sum += (ratings['rating'][i] - model.predict([np.array([ratings['user_id'
    ][i]]), np.array([ratings['movie_id'][i]])])) ** 2
mse = math.sqrt(sum/ratings.shape[0])
print(mse)
```

训练数据的误差在 0.8226 左右，大概一个评分等级不到的误差。

读者可能会问，为什么这个误差和之前矩阵分解的浅层模型误差的差距比较大？作者的理解是，这里的 Dropout 正则项起了很大的作用。虽然我们建了深层网络，但是由于有了 Dropout 这个正则项，必然会造成训练数据的信息丢失（这种丢失会让我们在测试数据时受益）。就好比加了 L_1, L_2 之类的正则项以后，估计的参数就不是无偏的了。因此，Dropout 是训练误差增加的原因，这是设计模型的必然结果。但是，需要记住的是，我们始终要对测试集上的预测做评估，训练集的误差只是看优化方向和算法是否大致有效。

5.4 其他常用算法

由于篇幅限制，这里只简单介绍推荐系统的其他常用算法。

协同过滤

协同过滤的含义是，利用众人的数据协助推断。一个经典的例子是，很多人买了牛奶的同时都买了面包，已知你买了牛奶，那么给你推荐面包就是很自然的事情。在实际数据上，这种方法效果一般，原因是类似于亚马逊等网站的商品太多了，用户之间很少能找到有很多重复的商品项，所以相似用户的构造会不准确。因而类似的规则便有很多噪声。

协同过滤示意图如图5.8所示。

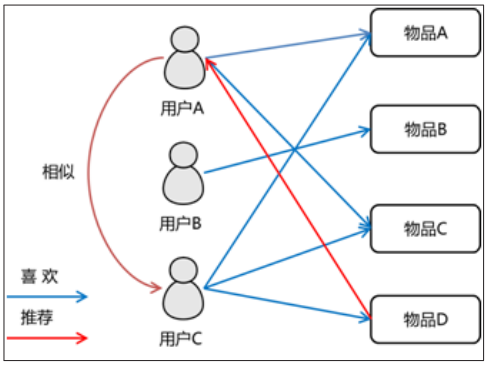


图 5.8　协同过滤示意图（图片来源：Google Image）

因子分解机

因子分解机是谷歌研究科学家 S. Rendle 教授提出的。它是矩阵分解的推广，可以使用多维特征变量。这个模型从回归的角度解释了因变量和自变量之间的联系，有显式表达式。求解也有交替最小二乘、蒙特卡洛模拟等算法支持，是一个非常强大的普适性模型（见图5.9）。该模型还证明了它是很多其他模型的特例，比如 SVD++ 等。具体看文章 *Factorization Machines*，发表在 *ICDM '10 Proceedings of the 2010 IEEE International Conference on Data Mining* 上。

$$\hat{y} := w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \langle \boldsymbol{v}_i, \boldsymbol{v}_j \rangle x_i x_j$$

图 5.9　因子分解机模型

玻尔兹曼向量机

玻尔兹曼向量机是谷歌副总裁、深度学习的开山鼻祖 Geoffrey Hinton 提出的。该模型建立了电影及其表征之间的概率联系。从用户的行为可以推断出用户对于电影表征

的偏好的概率表示；反过来，这些电影表征的偏好又可以用来给用户推荐电影。这种概率联系是通过 RBM 模型学出来的。这项技术发表在 *Proceedings of the 24th International Conference on Machine Learning* 上。玻尔兹曼向量机示意图如图5.10所示。

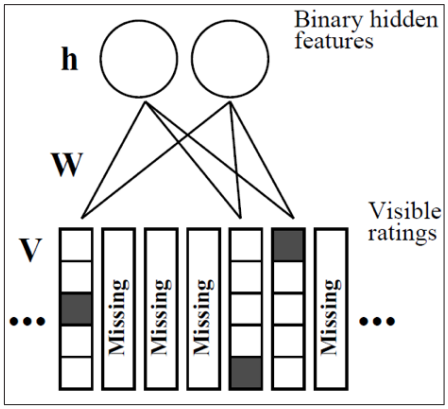


图 5.10 玻尔兹曼向量机示意图（图片来源：Google Image）

总的来说，推荐系统的算法层出不穷，也在商业上证明了自身的价值。上面介绍的各种模型在任何评分类或者具备隐含回馈数据（浏览、点击等）的问题上都可以使用，只是需要读者根据具体的业务情况对模型算法进行适当的改造。

5.5 评判模型指标

最后，我们简单讨论一下如何评判模型。评判模型一般有两种指标：线上和线下。

线上需要设计实验，基于一定的随机规则对用户、设备或者浏览器 Cookie 进行分组，然后设定一些指标，观察这些指标在实验期运用新模型是否比旧模型好。如果结论在统计意义上是肯定的，那么可以逐步把新模型运用到更大的群体中实验，最终百分之百上线。线上实验的指标设定需要快速收集到，因而必须是短期指标，比如点击率、转化率、购买率、访问量等。除此之外，我们应该用流数据的形式将这些数据导入到实验平台，这样可以更快地看到指标，从而做决定要不要进一步推广新的模型。

线上指标的优点是快速和因果关系明确；缺点是无法测试对长期目标的影响，并且容易不太稳定，受比如新奇效果（Novelty Effect）或者实验渗透率的影响。

我们对线下指标的要求要宽松很多，不但短期目标可以计算，长期目标也可以计算，比如留存率等。线下指标的缺点是因果关系不明确，额外因素有可能会干扰结果。通常的做法是，在线下建模型的时候，用线下指标给出一个最好的模型。然后，把这个新模型和现有的在线模型拿到线上去，进行数据收集和统计分析，利用短期指标给出是否要推广新模型的结论。

在推荐系统中，评分类的数据一般用均方差（Mean Squared Error）作为评判标准；而对于隐含回馈数据，一般用基于信息检索的概念中的精确率（推荐 10 部电影，用户看了几部）和召回率（用户感兴趣的 5 部电影，是否都在推荐列表里）作为最常用的指标。通常我们只能近似利用已有的历史数据去估计。比如在线下模型中，我们不知道用户没看是因为不知道还是不喜欢看，因为当时用户看到的推荐列表并不等同于新模型给出的推荐列表，当然这些也可以通过机器日志区分开。同样，我们也不可能知道用户感兴趣的所有电影。在线上试验中，点击率等就是比较好的指标，因为推荐是实时的，用户点没点跟他看到的推荐列表有直接的关系。最后提一下，在隐含回馈数据中，还可以结合使用用户观看进度的指标，比如之前提到的平均百分位数（Mean Percentile Rank）等。系统认为，如果推荐的内容被用户点了，并且花时间看了很大一部分，那么推荐就是有效的。这也是非常符合常理的思考。

6 图像识别

6.1 图像识别入门

图像识别是深度学习最典型的应用之一。关于深度学习的图像识别可以追溯很长的历史，其中最具有代表性的例子是手写字体识别和图片识别。手写字体识别主要是用机器正确区别手写体数字 0~9。银行支票上的手写体识别技术就是基于这个技术。图片识别的代表作就是 ImageNet。这个比赛需要团队识别图片中的动物或者物体，把它们正确地分到一千个类别中的其中一个。

图6.1和图6.2是 ImageNet 中的两个训练例子。它们都表示猫，但是猫的动作姿势各不相同。如何从图片中提取猫的特征，并且在变换图片（平移、旋转、放缩等）时，让机器仍然认为是猫，这是一个非常有挑战性的任务。



图 6.1　猫（图片来源：https:github.comBVLCcaffe）



图 6.2　猫（图片来源：http://www.image-net.orgsearch?q=cat）

6.2　卷积神经网络的介绍

图像识别有很多种技术可以实现，目前最主流的技术是深度神经网络，其中尤以卷积神经网络最为出名。卷积神经网络（见图6.3）是一种自动化特征提取的机器学习模型。从数学的角度看，任何一张图片都可以对应到 $224 \times 224 \times 3$ 或者 $32 \times 32 \times 3$ 等三维向量，这取决于像素。我们的目标是把这个三维向量（又被称为张量）映射到 N 个类别中的一类。神经网络就是建立了这样一个映射关系，或者称为函数。它通过建立网状结构，辅以矩阵的加、乘等运算，最后输出每个图像属于每个类别的概率，并且取概率最高的作为我们的决策依据。

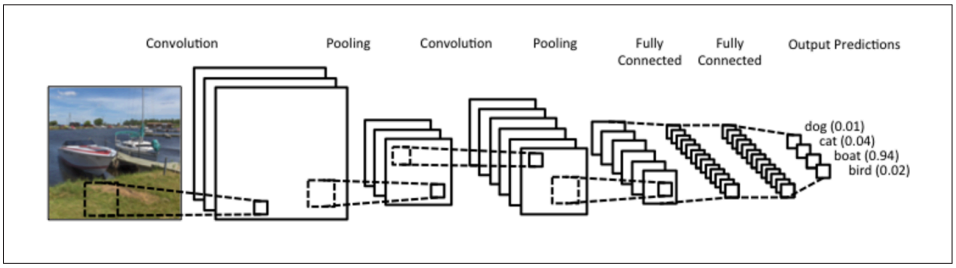


图 6.3　卷积神经网络（图片来源：http:www.wildml.com）

用深度学习解决图像识别的问题，从直观上讲，是一个从细节到抽象的过程。假如给定一张图，大脑中最先反应的是点和边，然后是由点和边抽象成各种形状，比如圆形、矩形、十字形等，之后抽象成脸、耳朵之类的特征，最后由这些特征决定图像到底属于哪类。比如脸扁圆，耳朵在头的两侧并成 45 度夹角和其他一些特征决定了这是猫还是狗，或者是其他动物。这里的关键是抽象。那么抽象是什么呢？抽象就是把图像中的各种零散的特征通过某种方式汇总起来，形成新的特征，而利用这些新的特征更容易

区分图像的类别。通过这种有监督的学习，分类（classification）任务能借助这些抽取出来的更具备区分作用的特征来更好地完成。深度神经网络最上层的特征是最抽象的。

抽象的核心是建立特征，或者叫特征工程。在传统的特征工程里，我们定义了一个叫过滤器（filter）的工具。过滤器带有特征指示，比如十字型过滤器等，它用来探测图像中是否具有十字型特征和图像中哪里具有十字型特征。十字型过滤器的作用就是对图像的局部像素进行卷积运算，使得过滤后的新图像在原图像中具有十字型特征的地方信号更强，在原图像中不具备十字型特征的地方信号更弱。这是一个基于过滤器的“去噪存真”的过程。我们通常会先构造一系列事先定义好的过滤器，然后从左到右挨个扫描图片的各个部分。这样每个过滤器会产生一个过滤后的图像，而这个图像又可以把是否具有过滤器提示的形状和哪里有这个形状表示出来，这样就起到了抽象的作用。通过这些抽象，再加以一些分类方法，比如支持向量机（SVM）等完成分类任务。这里的挑战是要大量地尝试和构造各种过滤器。

下面先看一个卷积神经网络中的过滤器的例子：过滤器扫描 RGB 图像，每次扫描一个局部，这样返回一个平面。当有多个过滤器作用的时候，这些平面就可以叠加，形成三维立体状。扫描 RGB 图像用的过滤器一般是三维的，比如 (5, 5, 3)，总共有 75 个参数。过滤器示意图如图6.4所示。

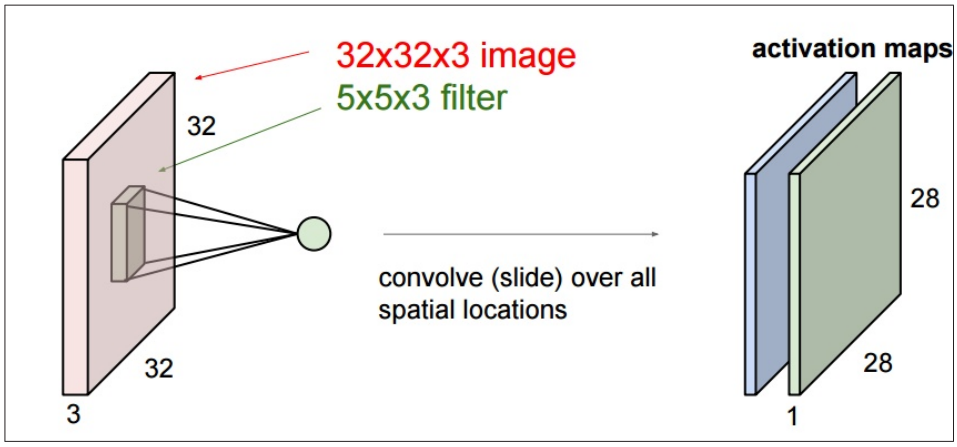


图 6.4 过滤器（图片来源：Google Image）

卷积神经网络的威力在于其可以自动学习过滤器。为什么它可以自动学习呢？主要是因为卷积神经网络有反馈机制。这决定了以下几点：第一，过滤器必须对分类有帮助。不能随便定义过滤器，如果随便定义过滤器，那么它就不能有效地分类，准确率会很不理想。第二，网络有调整机制。如果将任务分给了一个随机的过滤器，分类不行怎么办？系统会知道应该在哪里调过滤器的权重，往什么方向调，和各层网络之间的权重怎么调整等。每调一次，系统对于分类任务就会更精确一些，于是经过上百次、上千

次，甚至更多次的迭代，最终模型会越来越精确。这就是著名的后向传播算法。后向传播算法的本质是高等代数里的链式法则。其原理就是机器不断通过现有参数在批量数据上所得到的标注和这些批量数据的真实标注的差距，给网络指示怎么调整网络模型和过滤器，即各种参数，从而在下一次批量数据上表现更好一些。下一次批量数据经过调整后的模型可能仍有很大的误差，网络会提示怎么进一步调整模型的权重和过滤器。这样不断反复，最终等过滤器和网络权重稳定下来，网络的训练就完成了。这是最基本的卷积神经网络的算法，当然在实践过程中，还会有各种其他处理方式，比如正则化等，在后面会提到。

卷积神经网络是深度学习的一种模型。它和一般的深度学习模型的主要区别是对模型有两个强假设。一般的深度学习模型只是假设模型有几层，每层有几个节点，然后把上下层之间的节点全部连接起来，这种模型的优点是灵活，缺点是灵活带来的副作用，即过度拟合。因为模型的参数太多，会把训练数据的噪音也模拟进去，从而让模型的普适性大打折扣。卷积神经网络和包括循环神经网络、长短记忆网络等其他模型之所以更流行，就是因为它们对模型有两个强假设，而这些强假设在某些特定任务中是合理的，比如卷积神经网络用于图像识别，循环神经网络和长短记忆网络用于自然语言处理任务。

这里提到的两个强假设，第一个是参数共享。过滤器一般需要的参数比较少，比如 $5 \times 5 \times 3$ 的过滤器需要 75 个参数就可以了。这和多层神经网络相比，相当于我们只是把隐含层和局部输入联系在一起，而这两层之间的权重只需要 75 个参数。其他超过这个局部范围的区域的网络权重都是 0。当然只有一个过滤器是不够的，我们需要构造多个过滤器，不过总体来说省了很多参数。第二个是局部像素的相关性，即局部区域的像素值一般差的不太多。基于此衍生了一种处理技术——Max Pooling，即在局部，比如在 224×224 的格子里，取局部区域像素值的最大值。在图像中，局部像素值之间是有相关性的，所以取局部最大像素值并没有损失很多信息。进行 Max Pooling 之后得到的图像维度以平方比的速度缩小。这个简单的假设大大节省了后续参数。

前面提到，卷积神经网络利用过滤器遍历图像进行局部扫描，而全连接神经网络可以被认为是全局扫描。这么一来，卷积神经网络和全连接神经网络的关系自然就是辩证统一的了：对一个 $224 \times 224 \times 3$ 的图像利用 1000 个长相为 $224 \times 224 \times 3$ 的过滤器进行扫描，这就回到了我们熟悉的全连接神经网络模型。这里的过滤器扫描的“局部区域”即“全部区域”。可以这么理解：每个类都对应着一个过滤器，给定了 1000 个过滤器以后，这 1000 个类别所对应的值也就给定了，最后进行分类时取 1000 个数值中的最大值就可以了。这里的过滤器等价于全连接神经网络的每个输出节点和所有输入层节点的权重。

从另一个角度看，任何一个卷积神经网络其实都是通过对整个神经网络权重附加

参数共享这一限制而实现的。所以卷积神经网络和全连接神经网络是可以相互转换的。

一般一个卷积层包括 3 个部分：卷积步骤、非线性变化（一般是 relu、tanh、sigmoid 等）和 Max Pooling。有的网络还包括了 Dropout 这一步。总结来说，一些流行的卷积神经网络，比如 LeNet、VGG16 等，都是通过构造多层的卷积层，使得原来“矮胖”型的图像输入层（224 × 224 × 3）立体，变成比如 1 × 1 × 4096 之类的“瘦长”型立体，最后做一个单层的网络，把“瘦长”型立体和输出层（类别）联系在一起。

下面介绍几个流行的卷积神经网络。

(1) AlexNet，如图6.5所示，来源于 *ImageNet Classification with Deep Convolutional Neural Networks*, NIPS 2012 这篇文章。

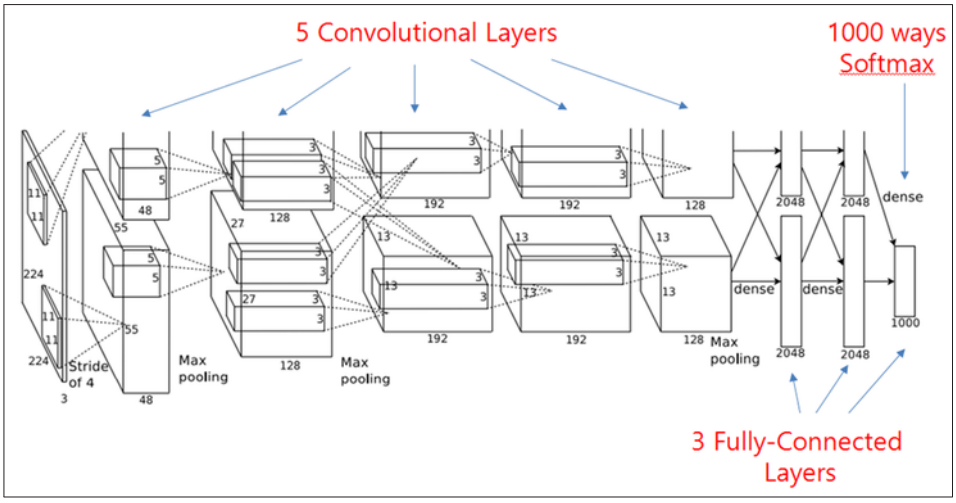


图 6.5　AlexNet（图片来源：https:world4jason.gitbooks.ioresearch-log）

(2) LeNet，如图6.6所示。关于此网络结构，可参见 *Gradient-based learning applied to document recognition*. Proceedings of the IEEE, ovember 1998 这篇文章。

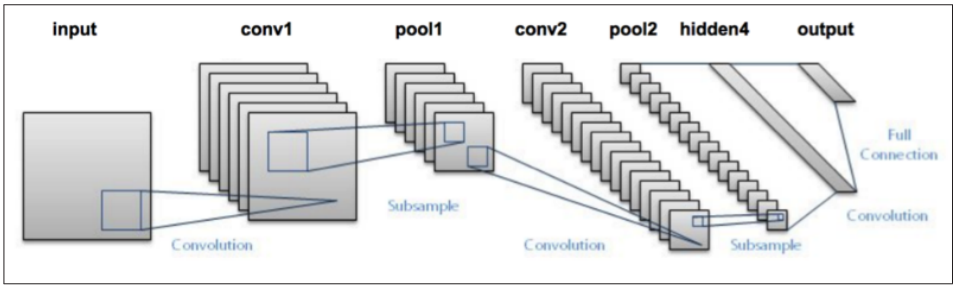


图 6.6　LeNet（图片来源：http:www.pyimagesearch.com）

(3) VGG16，如图6.7所示。关于此网络结构，可参见 *Very Deep Convolutional Networks for Large-Scale Image Recognition* 这篇文章。

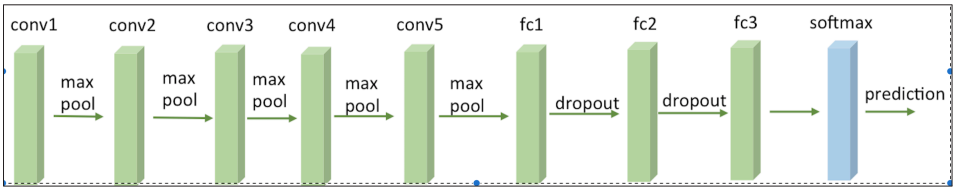


图 6.7　VGG16（图片来源：http:blog.christianperone.com）

(4) VGG19，如图6.8所示。参考资料同上。

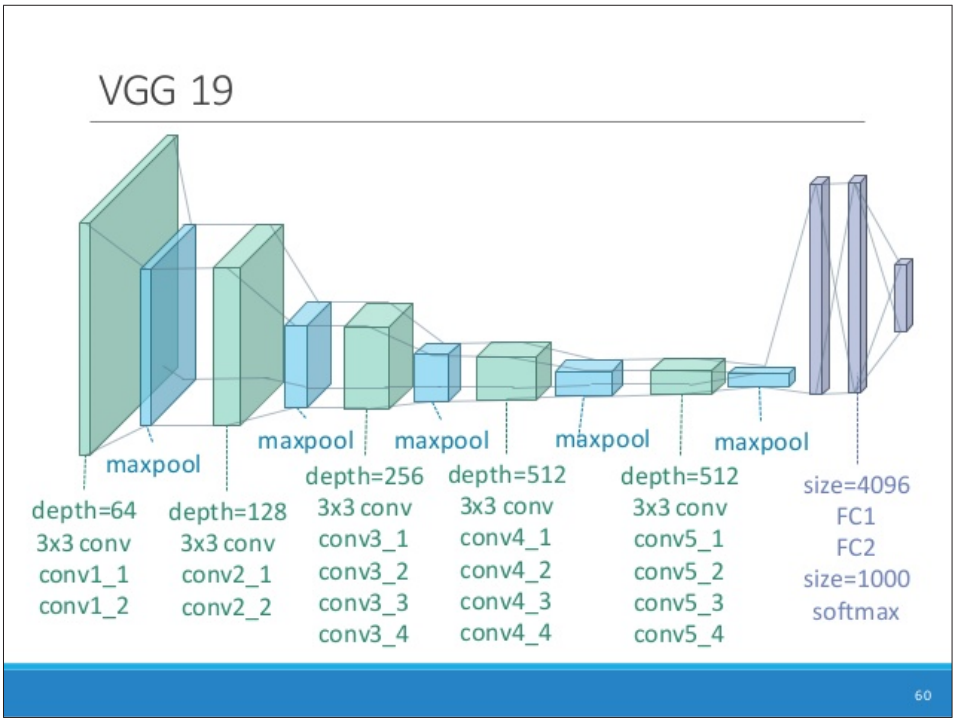


图 6.8　VGG19（图片来源: https:www.slideshare.net/ckmarkohchang）

关于卷积神经网络还要补充两点内容。第一，在局部扫描的过程中，有一个参数叫步长，就是指过滤器以多大的跨度上下或左右平移地扫描。第二，对于经由过滤器局部扫描后的卷积层图像，由于处理边界不同，一般有两种处理方式。一种是在局部扫描过程中对图像边界以外的一层或多层填上 0，平移的时候可以将其移出边界到达 0 的区域。这样的好处是在以 1 为步长的局部扫描完以后，所得的新图像和原图像长宽一致，被称作 zero padding(same padding)。另一种是不对边界外做任何 0 的假定，所有平移都在边界内，被称作 valid padding，使用这种方式通常扫描完的图像尺寸会比原来的小。

图6.9比较形象地比较两者的区别。左图是 same padding(zero padding)，右图是 valid padding。

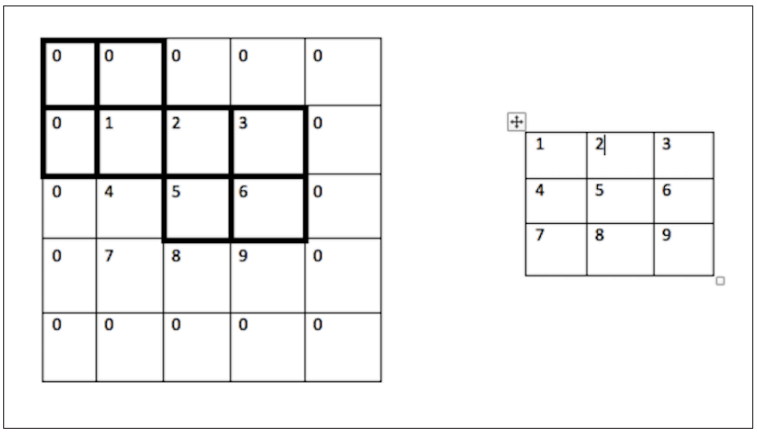


图 6.9　填充选择

关于理论部分先介绍到这里，接下来介绍两个实例。第一个例子是利用 MNIST 字体数据库构造卷积神经网络，用来识别手写体数字。这里会演示如何做一套端到端的深度学习系统。第二个例子是用 VGG16 模型作为模型框架处理同样的字体识别问题。这类建模方法被称作迁移学习，即利用别人的模型作为自己模型的输入，或者作为自己问题中的已知部分。这种学习是站在别人的肩膀上，从而大大缩短自己调整模型和建立模型的时间。同时，也可以选择利用已经建好模型的参数值，再适当地加上少量的参数，最后只需要计算自己添加的那部分参数的值就可以了。

6.3　端到端的 MNIST 训练数字识别

下面介绍端到端的 MNIST 训练数字识别过程。

这个数据集是由 LeCun Yang 教授和他团队整理的，囊括了 6 万个训练集和 1 万个测试集。每个样本都是 32 × 32 的像素值，并且是黑白的，没有 R、G、B 三层。我们

要做的是把每个图片分到 0~9 类别中。

图6.10是一些手写数字的样本。

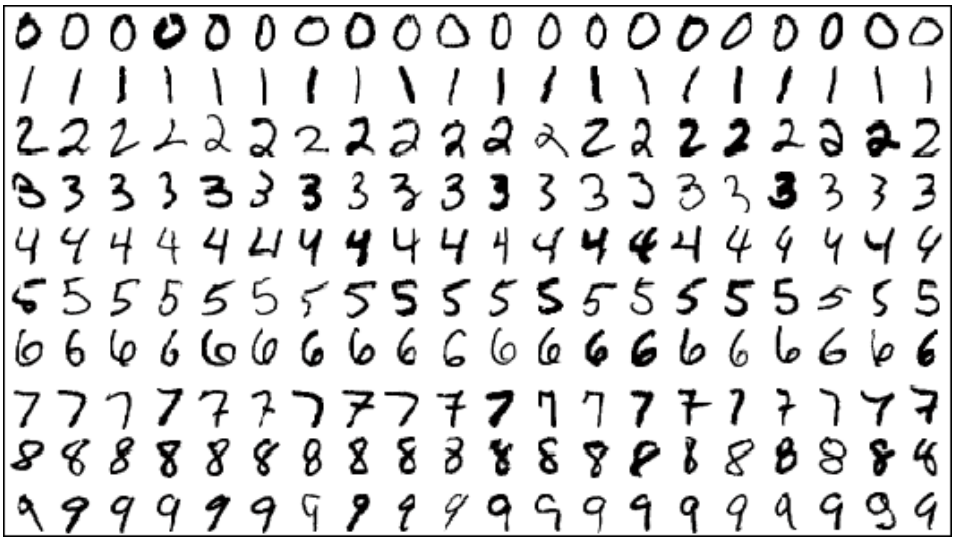


图 6.10　MNIST 字体样本（图片来源：http:myselph.deneuralNet.html ）

接下来用 Keras 搭建卷积网络训练模型。幸运的是，Keras 自带了训练和测试数据集。数据格式都已经整理完毕，我们所要做的就是搭建 Keras 模块，并且确保训练集和测试集的数据和模块的参数相吻合。

```
import numpy as np
from keras.datasets import mnist
```

引入 Keras 的卷积模块，包括 Dropout、Conv2D 和 MaxPooling2D。

```
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers.convolutional import Conv2D, MaxPooling2D
```

先读入数据：

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
```

看一下数据集长什么样子：

```
print(X_train[0].shape)
print(y_train[0])
```

结果分别显示 (28, 28) 和 5。由此，训练数据集图像是 28 × 28 的格式，而标签类别是 0~9 的数字。下面把训练集中的手写黑白字体变成标准的四维张量形式，即（样本数量，长，宽，1），并把像素值变成浮点格式。

```
X_train = X_train.reshape(X_train.shape[0],28,28,1).astype('float32')
X_test = X_test.reshape(X_test.shape[0],28,28,1).astype('float32')
```

由于每个像素值都是介于 0~255，所以这里统一除以 255，把像素值控制在 0~1 范围。

```
X_train /= 255
X_test /= 255
```

由于输入层需要 10 个节点，所以最好把目标数字 0~9 做成 One Hot 编码的形式。

```
def tran_y(y):
    y_ohe = np.zeros(10)
    y_ohe[y] = 1
    return y_ohe
```

把标签用 One Hot 编码重新表示一下。

```
y_train_ohe = np.array([tran_y(y_train[i]) for i in range(len(y_train))])
y_test_ohe = np.array([tran_y(y_test[i]) for i in range(len(y_test))])
```

接着搭建卷积神经网络。

```
model = Sequential()
```

添加一层卷积层，构造 64 个过滤器，每个过滤器覆盖范围是 3 × 3 × 1。过滤器挪动步长为 1，图像四周补一圈 0，并用 relu 进行非线性变换。

```
model.add(Conv2D(filters = 64, kernel_size = (3, 3), strides = (1, 1),
padding = 'same', input_shape = (28,28,1), activation = 'relu'))
```

添加一层 Max Pooling，在 2 × 2 的格子中取最大值。

```
model.add(MaxPooling2D(pool_size = (2, 2)))
```

设立 Dropout 层。将 Dropout 的概率设为 0.5。读者也可以尝试设为 0.2 或 0.3 这些常用的值。

```
model.add(Dropout(0.5))
```

重复构造，搭建深度网络。

```
model.add(Conv2D(128, kernel_size = (3, 3), strides = (1, 1), padding = '
same', activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))
model.add(Conv2D(256, kernel_size = (3, 3), strides = (1, 1), padding = '
same', activation = 'relu'))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Dropout(0.5))
```

把当前层节点展平。

```
model.add(Flatten())
```

构造全连接神经网络层。

```
model.add(Dense(128, activation = 'relu'))
model.add(Dense(64, activation = 'relu'))
model.add(Dense(32, activation = 'relu'))
model.add(Dense(10, activation = 'softmax'))
```

最后定义损失函数，一般来说分类问题的损失函数都选择采用交叉熵（Cross Entropy）。

```
model.compile(loss = 'categorical_crossentropy', optimizer = 'adagrad',
metrics = ['accuracy'])
```

放入批量样本，进行训练。

```
model.fit(X_train, y_train_ohe, validation_data = (X_test, y_test_ohe),
epochs = 20, batch_size = 128)
```

在测试集上评价模型的准确度：

```
scores = model.evaluate(X_test, y_test_ohe, verbose = 0)
```

最后获得的精确度为 99.4%。

6.4 利用 VGG16 网络进行字体识别

接下来使用迁移学习的思想，以 VGG16 作为模板搭建模型，训练识别手写字体。

VGG16 模型是基于 K. Simonyan 和 A. Zisserman 写的 Very Deep Convolutional Networks for Large-Scale Image Recognition, arXiv:1409.1556。

首先引入 Keras 里的 VGG16 模块。

```
from keras.applications.vgg16 import VGG16
```

其次加载 Keras 模型：

```
from keras.layers import Input, Flatten, Dense, Dropout
from keras.models import Model
from keras.optimizers import SGD
```

加载字体库作为训练样本。如果是第一次加载，则 Keras 会从 AWS 的存储账号下载数据。

```
from keras.datasets import mnist
```

加载 OpenCV（在命令行窗口中输入 pip install opencv-python），这里为了后期对图像的处理，比如尺寸变换和 Channel 变换。这些变换是为了使图像满足 VGG16 所需要的输入格式。

```
import cv2
import h5py as h5py
import numpy as np
```

接下来新建一个模型，其类型是 Keras 的 Model 类对象。我们构建的模型会将 VGG16 顶层去掉，只保留其余的网络结构。这里用 include_top = False 表明我们迁移除顶层以外的其余网络结构到自己的模型中。

```
model_vgg = VGG16(include_top = False, weights = 'imagenet', input_shape =
(224,224,3))
model = Flatten(name = 'flatten')(model_vgg.output)
model = Dense(10, activation = 'softmax')(model)
model_vgg_mnist = Model(model_vgg.input, model, name = 'vgg16')
```

打印模型结构，包括所需要的参数。

```
model_vgg_mnist.summary()
```

```
_________________________________________________________
Layer (type)              Output Shape          Param #
=========================================================
input_10 (InputLayer)     (None, 224, 224, 3)   0
vgg16 (Model)             (None, 7, 7, 512)     14714688
flatten (Flatten)         (None, 25088)         0
dense_15 (Dense)          (None, 10)            250890
=========================================================
Total params: 14,965,578
Trainable params: 14,965,578
Non-trainable params: 0
```

在这里可以看到，所有 1496 万个网络权重（VGG16 网络权重加上我们搭建的权重）都需要训练，这是因为我们迁移了网络结构，但是没有迁移 VGG16 网络权重。迁移网络权重的好处在于网络权重不用重新训练，只需要训练最上层搭建的部分就行了。坏处是，新的数据不一定适用已训练好的权重，因为已训练好的权重是基于其他数据训练的，数据分布和我们关心的问题可能完全不一样。这里虽然引进了 VGG 在 ImageNet 中的结构，但是具体模型仍需要在 VGG16 的框架上加工。

另外，本地机器很有可能不能把整个模型和数据放入内存进行训练，出现 Kill:9 的内存不够的错误。如果想要训练，则建议用较少样本，或者把样本批量减小至比如 32。有条件的话可以用 AWS 里的 EC2 GPU Instance g2.2xlarge/g2.8xlarge 进行训练。

作为对比，我们建立另外一个模型，这个模型的特点是把 VGG16 网络的结构和权重同时迁移。这里的关键点是把不需要重新训练的权重“冷冻”起来。这里使用 `trainable = false` 这个选项。注意，这里我们定义输入的维度为（224,224,3），因此需要较大的内存，除非读者使用数据生成器迭代对象。如果内存较小，那么可以将维度降为（112,112,3），这样在 32GB 内存的机器上也能顺利运行。

```
ishape=224
model_vgg = VGG16(include_top = False, weights = 'imagenet', input_shape = (
ishape, ishape, 3))
for layer in model_vgg.layers:
    layer.trainable = False
model = Flatten()(model_vgg.output)
model = Dense(10, activation = 'softmax')(model)
```

```
model_vgg_mnist_pretrain = Model(model_vgg.input, model, name = '
vgg16_pretrain')
```

打印模型结构，包括所需要的参数。

```
model_vgg_mnist_pretrain.summary()
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_11 (InputLayer)        (None, 224, 224, 3)       0
block1_conv1 (Conv2D)        (None, 224, 224, 64)      1792
block1_conv2 (Conv2D)        (None, 224, 224, 64)      36928
block1_pool (MaxPooling2D)   (None, 112, 112, 64)      0
block2_conv1 (Conv2D)        (None, 112, 112, 128)     73856
block2_conv2 (Conv2D)        (None, 112, 112, 128)     147584
block2_pool (MaxPooling2D)   (None, 56, 56, 128)       0
block3_conv1 (Conv2D)        (None, 56, 56, 256)       295168
block3_conv2 (Conv2D)        (None, 56, 56, 256)       590080
block3_conv3 (Conv2D)        (None, 56, 56, 256)       590080
block3_pool (MaxPooling2D)   (None, 28, 28, 256)       0
block4_conv1 (Conv2D)        (None, 28, 28, 512)       1180160
block4_conv2 (Conv2D)        (None, 28, 28, 512)       2359808
block4_conv3 (Conv2D)        (None, 28, 28, 512)       2359808
block4_pool (MaxPooling2D)   (None, 14, 14, 512)       0
block5_conv1 (Conv2D)        (None, 14, 14, 512)       2359808
block5_conv2 (Conv2D)        (None, 14, 14, 512)       2359808
block5_conv3 (Conv2D)        (None, 14, 14, 512)       2359808
block5_pool (MaxPooling2D)   (None, 7, 7, 512)         0
flatten_2 (Flatten)          (None, 25088)             0
dense_16 (Dense)             (None, 10)                250890
=================================================================
Total params: 14,965,578
Trainable params: 250,890
Non-trainable params: 14,714,688
```

我们只需要训练 25 万个参数，比之前整整少了 60 倍！

```
sgd = SGD(lr = 0.05, decay = 1e-5)
```

```
model_vgg_mnist_pretrain.compile(loss = 'categorical_crossentropy',
optimizer = sgd, metrics = ['accuracy'])
```

因为 VGG16 网络对输入层的要求，我们用 OpenCV 把图像从 32 × 32 变成 224 × 224（cv2.resize 的命令），把黑白图像转换为 RGB 图像（cv2.COLOR_GRAY2BGR），并且把训练数据转化成张量形式，供 Keras 输入。

```
(X_train, y_train), (X_test, y_test) = mnist.load_data()
X_train = [cv2.cvtColor(cv2.resize(i, (ishape, ishape)), cv2.COLOR_GRAY2BGR)
 for i in X_train]
X_train = np.concatenate([arr[np.newaxis] for arr in X_train]).astype('
float32')
X_test = [cv2.cvtColor(cv2.resize(i, (ishape, ishape)), cv2.COLOR_GRAY2BGR)
for i in X_test]
X_test = np.concatenate([arr[np.newaxis] for arr in X_test]).astype('float32
')
```

训练数据的维度如下，我们有 6 万个样本，每个样本是 224 × 224 × 3 的张量。

```
X_train.shape
(60000, 224, 224, 3)
X_test.shape
(10000, 224, 224, 3)
```

```
X_train = X_train/255
X_test = X_test/255
```

看一看训练数据是否有数据丢失，查找非零项后，看上去没问题。

```
np.where(X_train[0]!= 0)
(array([ 36, 36, 36, ..., 203, 203, 203]), array([103, 103, 103, ..., 95,
95, 95]), array([0, 1, 2, ..., 0, 1, 2]))
```

至此，训练数据集和测试数据集的图像部分已经完成。

最后，把训练数据集和测试数据集的类别属性（0~9）转换成 One Hot 编码形式，作为输出层的维度。

```
def tran_y(y):
    y_ohe = np.zeros(10)
```

```
        y_ohe[y] = 1
    return y_ohe
```

```
y_train_ohe = np.array([tran_y(y_train[i]) for i in range(len(y_train))])
y_test_ohe = np.array([tran_y(y_test[i]) for i in range(len(y_test))])
```

在对 MINST 数据集进行训练。

```
model_vgg_mnist_pretrain.fit(X_train, y_train_ohe, validation_data = (X_test
, y_test_ohe), epochs = 200, batch_size = 128)
```

6.5 总结

本章回顾了卷积神经网络，并且介绍了用 Keras 建立端到端的卷积神经网络，用于进行手写字体的分类。同时本章也回顾了几个经典的卷积神经网络，并利用迁移学习在 VGG16 模型的基础上进行加工，构造深度学习网络，进行字体识别训练。这两种建模思路各有千秋，读者可以举一反三，从而对卷积神经网络、深度学习和迁移学习有更深的理解和应用。

7 自然语言情感分析

7.1 自然语言情感分析简介

情感分析无处不在，它是一种基于自然语言处理的分类技术。其主要解决的问题是给定一段话，判断这段话是正面的还是负面的。例如在亚马逊网站或者推特网站中，人们会发表评论，谈论某个商品、事件或人物。商家可以利用情感分析工具知道用户对自己的产品的使用体验和评价。当需要大规模的情感分析时，肉眼的处理能力就变得十分有限了。情感分析的本质就是根据已知的文字和情感符号，推测文字是正面的还是负面的。处理好了情感分析，可以大大提升人们对于事物的理解效率，也可以利用情感分析的结论为其他人或事物服务，比如不少基金公司利用人们对于某家公司、某个行业、某件事情的看法态度来预测未来股票的涨跌。

进行情感分析有如下难点：第一，文字非结构化，有长有短，很难适合经典的机器学习分类模型。第二，特征不容易提取。文字可能是谈论这个主题的，也可能是谈论人物、商品或事件的。人工提取特征耗费的精力太大，效果也不好。第三，词与词之间有联系，把这部分信息纳入模型中也不容易。

本章探讨深度学习在情感分析中的应用。深度学习适合做文字处理和语义理解，是因为深度学习结构灵活，其底层利用词嵌入技术可以避免文字长短不均带来的处理困难。使用深度学习抽象特征，可以避免大量人工提取特征的工作。深度学习可以模拟词与词之间的联系，有局部特征抽象化和记忆功能。正是这几个优势，使得深度学习在情感分析，乃至文本分析理解中发挥着举足轻重的作用。

顺便说一句，推特已经公开了他们的情感分析 API（http://help.sentiment140.com/api）。读者可以把其整合到自己的应用程序中，也可以试着开发一套自己的 API。下面通过一个电影评论的例子详细讲解深度学习在情感分析中的关键技术。

首先下载 http://ai.stanford.edu/~amaas/data/sentiment/中的数据。

输入下文安装必要的软件包：

```
pip install numpy scipy
pip install scikit-learn
pip install pillow
pip install h5py
```

下面处理数据。Keras 自带了 imdb 的数据和调取数据的函数，直接调用 load.data() 就可以了。

```
import keras
import numpy as np
from keras.datasets import imdb
(X_train, y_train), (X_test, y_test) = imdb.load_data()
```

先看一看数据长什么样子的。输入命令：

```
X_train[0]
```

我们可以看到结果：

```
array([[    1,    14,    22,    16,    43,   530,   973,  1622,  1385,
          65,   458,  4468,    66,  3941,     4,   173,    36,   256,
           5,    25,   100,    43,   838,   112,    50,   670, 22665,
           9,    35,   480,   284,     5,   150,     4,   172,   112,
         167, 21631,   336,   385,    39,     4,   172,  4536,  1111,
          17,   546,    38,    13,   447,     4,   192,    50,    16,
           6,   147,  2025,    19,    14,    22,     4,  1920,  4613,
         469,     4,    22,    71,    87,    12,    16,    43,   530,
          38,    76,    15,    13,  1247,     4,    22,    17,   515,
          17,    12,    16,   626,    18, 19193,     5,    62,   386,
          12,     8,   316,     8,   106,     5,     4,  2223,  5244,
          16,   480,    66,  3785,    33,     4,   130,    12,    16,
          38,   619,     5,    25,   124,    51,    36,   135,    48,
          25,  1415,    33,     6,    22,    12,   215,    28,    77,
```

```
15              52,     5,    14,   407,    16,    82, 10311,     8,     4,
16             107,   117,  5952,    15,   256,     4, 31050,     7,  3766,
17               5,   723,    36,    71,    43,   530,   476,    26,   400,
18             317,    46,     7,     4, 12118,  1029,    13,   104,    88,
19               4,   381,    15,   297,    98,    32,  2071,    56,    26,
20             141,     6,   194,  7486,    18,     4,   226,    22,    21,
21             134,   476,    26,   480,     5,   144,    30,  5535,    18,
22              51,    36,    28,   224,    92,    25,   104,     4,   226,
23              65,    16,    38,  1334,    88,    12,    16,   283,     5,
24              16,  4472,   113,   103,    32,    15,    16,  5345,    19,
25             178,    32]])
```

原来，Keras 自带的 load_data 函数帮我们从亚马逊 S3 中下载了数据，并且给每个词标注了一个索引 (index)，创建了字典。每段文字的每个词对应了一个数字。

```
print(y[:10])
```

得到 array([1, 0, 0, 1, 0, 0, 1, 0, 1, 0])，可见 y 就是标注，1 表示正面，0 表示负面。

```
1 print(X_train.shape)
2 print(y_train.shape)
```

我们得到的两个张量的维度都为 (25000,)。

接下来可以看一看平均每个评论有多少个字：

```
avg_len = list(map(len, X_train))


print(np.mean(avg_len))
```

可以看到平均字长为 238.714。

为了直观显示，这里画一个分布图（见图7.1）：

```
1 import matplotlib.pyplot as plt
2 plt.hist(avg_len, bins = range(min(avg_len), max(avg_len) + 50, 50))
3 plt.show()
```

注意，如果遇到其他类型的数据，或者自己有数据，那么就得自己写一套处理数据的脚本。大致步骤如下。

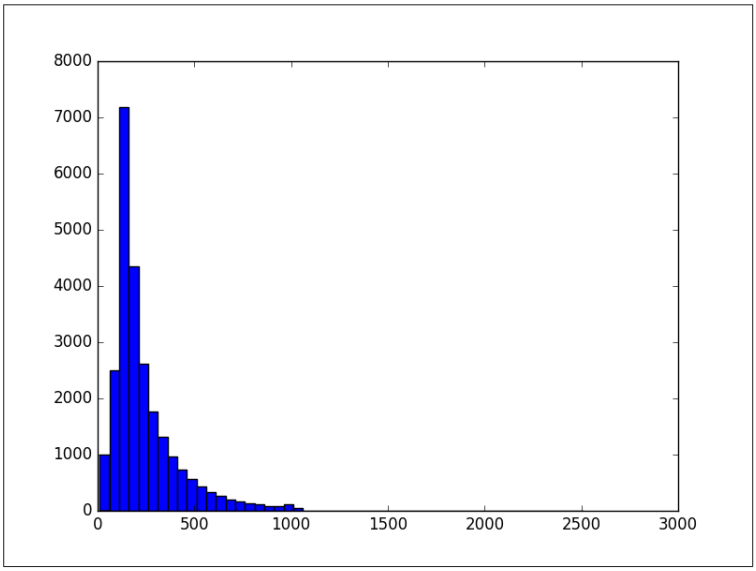


图 7.1　词频分布直方图

- 第一，文字分词。英语分词可以按照空格分词，中文分词可以参考 jieba。
- 第二，建立字典，给每个词标号。
- 第三，把段落按字典翻译成数字，变成一个 array。

接下来就开始建模了。

7.2　文字情感分析建模

7.2.1　词嵌入技术

为了克服文字长短不均和将词与词之间的联系纳入模型中的困难，人们使用了一种技术——词嵌入。简单说来，就是给每个词赋一个向量，向量代表空间里的点，含义接近的词，其向量也接近，这样对于词的操作就可以转化为对于向量的操作了，在深度学习中，这被叫作张量（tensor）。用张量表示词的好处在于：第一，可以克服文字长短不均的问题，因为如果每个词已经有对应的词向量，那么对于长度为 N 的文本，只要选取对应的 N 个词所代表的向量并按文本中词的先后顺序排在一起，就是输入张量了，其中每个词向量的维度都是一样的。第二，词本身无法形成特征，但是张量就是抽象的量化，它是通过多层神经网络的层层抽象计算出来的。第三，文本是由词组成的，

文本的特征可以由词的张量组合。文本的张量蕴含了多个词之间的组合含义，这可以被认为是文本的特征工程，进而为机器学习文本分类提供基础。

词的嵌入最经典的作品是 Word2Vec，可以参见：https://code.google.com/archive/p/word2vec/。通过对具有数十亿词的新闻文章进行训练，Google 提供了一组词向量的结果，可以从 http://word2vec.googlecode.com/svn/trunk/获取。其主要思想依然是把词表示成向量的形式，而不是 One Hot 编码。图7.2展示了这个模型里面词与词的关系。

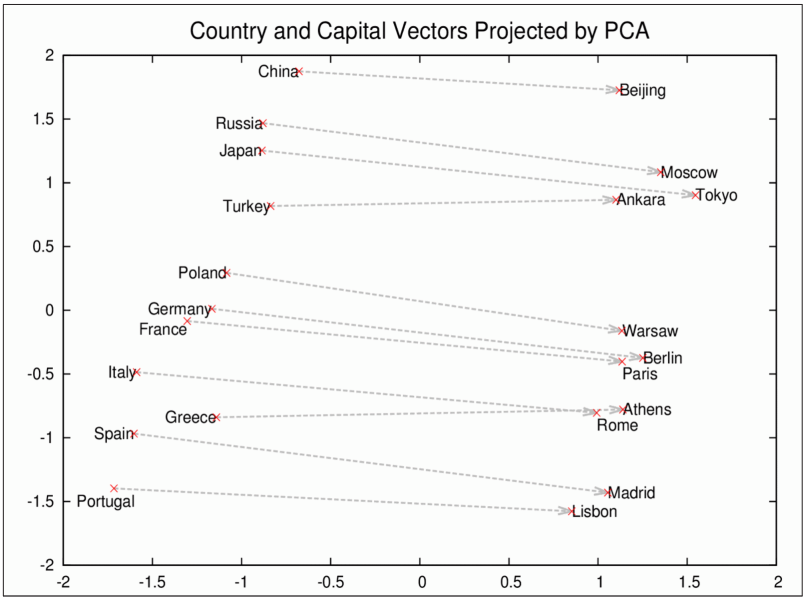


图 7.2　词向量示意图（图片来源：https://deeplearning4j.org/word2vec）

7.2.2　多层全连接神经网络训练情感分析

不同于已经训练好的词向量，Keras 提供了设计嵌入层（Embedding Layer）的模板。只要在建模的时候加一行 Embedding Layer 函数的代码就可以。注意，嵌入层一般是需要通过数据学习的，读者也可以借用已经训练好的嵌入层比如 Word2Vec 中预训练好的词向量直接放入模型，或者把预训练好的词向量作为嵌入层初始值，进行再训练。Embedding 函数定义了嵌入层的框架，其一般有 3 个变量：字典的长度（即文本中有多少词向量）、词向量的维度和每个文本输入的长度。注意，前文提到过每个文本可长可短，所以可以采用 Padding 技术取最长的文本长度作为文本的输入长度，而不足长度的都用空格填满，即把空格当成一个特殊字符处理。空格本身一般也会被赋予词向量，这可以通过机器学习训练出来。Keras 提供了 sequence.pad_sequences 函数帮我们做文本的处理和填充工作。

先把代码进行整理：

```
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.embeddings import Embedding
from keras.preprocessing import sequence
import keras
import numpy as np
from keras.datasets import imdb
(X_train, y_train), (X_test, y_test) = imdb.load_data()
```

使用下面的命令计算最长的文本长度：

```
m = max(list(map(len, X_train)), list(map(len, X_test)))
print(m)
```

从中我们会发现有一个文本特别长，居然有 2494 个字符。这种异常值需要排除，考虑到文本的平均长度为 230 个字符，可以设定最多输入的文本长度为 400 个字符，不足 400 个字符的文本用空格填充，超过 400 个字符的文本截取 400 个字符，Keras 默认截取后 400 个字符。

```
maxword = 400
X_train = sequence.pad_sequences(X_train, maxlen = maxword)
X_test = sequence.pad_sequences(X_test, maxlen = maxword)
vocab_size = np.max([np.max(X_train[i]) for i in range(X_train.shape[0])]) + 1
```

这里 1 代表空格，其索引被认为是 0。

下面先从最简单的多层神经网络开始尝试：

首先建立序列模型，逐步往上搭建网络。

```
model = Sequential()
model.add(Embedding(vocab_size, 64, input_length = maxword))
```

第一层是嵌入层，定义了嵌入层的矩阵为 vocab_size ×64。每个训练段落为其中的 maxword ×64 矩阵，作为数据的输入，填入输入层。

```
model.add(Flatten())
```

把输入层压平，原来是 maxword ×64 的矩阵，现在变成一维的长度为 maxword ×64 的向量。

接下来不断搭建全连接神经网络，使用 relu 函数。relu 是简单的非线性函数：$f(x) = \max(0, x)$。注意到神经网络的本质是把输入进行非线性变换。

```
model.add(Dense(2000, activation = 'relu'))
model.add(Dense(500, activation = 'relu'))
model.add(Dense(200, activation = 'relu'))
model.add(Dense(50, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
```

这里最后一层用 Sigmoid，预测 0，1 变量的概率，类似于 logistic regression 的链接函数，目的是把线性变成非线性，并把目标值控制在 0~1。因此这里计算的是最后输出的是 0 或者 1 的概率。

```
model.compile(loss = 'binary_crossentropy', optimizer = 'adam', metrics = ['
accuracy'])
print(model.summary())
```

这里有几个概念要提一下：交叉熵（Cross Entropy）和 Adam Optimizer。

交叉熵主要是衡量预测的 0，1 概率分布和实际的 0，1 值是不是匹配，交叉熵越小，说明匹配得越准确，模型精度越高。

其具体形式为

$$y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

这里把交叉熵作为目标函数。我们的目的是选择合适的模型，使这个目标函数在未知数据集上的平均值越低越好。所以，我们要看的是模型在测试数据（训练时需要被屏蔽）上的表现。

Adam Optimizer 是一种优化办法，目的是在模型训练中使用的梯度下降方法中，合理地动态选择学习速度（Learning Rate），也就是每步梯度下降的幅度。直观地说，如果在训练中损失函数接近最小值了，则每步梯度下降幅度自然需要减小，而如果损失函数的曲线还很陡，则下降幅度可以稍大一些。从优化的角度讲，深度学习网络还有其他一些梯度下降优化方法，比如 Adagrad 等。它们的本质都是解决在调整神经网络模型过程中如何控制学习速度的问题。

Keras 提供的建模 API 让我们既能训练数据，又能在验证数据时看到模型测试效果。

```
model.fit(X_train, y_train, validation_data = (X_test, y_test), epochs = 20,
 batch_size = 100, verbose = 1)
score = model.evaluate(X_test, y_test)
```

其精确度大约在85%。如果多做几次迭代，则精确度会更高。读者可以试着尝试一下多跑几个循环。

以上提到的是最常用的多层全连接神经网络模型。它假设模型中的所有上一层和下一层是互相连接的，是最广泛的模型。

7.2.3 卷积神经网络训练情感分析

全连接神经网络几乎对网络模型没有任何限制，但缺点是过度拟合，即拟合了过多噪声。全连接神经网络模型的特点是灵活、参数多。在实际应用中，我们可能会对模型加上一些限制，使其适合数据的特点。并且由于模型的限制，其参数会大幅减少。这降低了模型的复杂度，模型的普适性进而会提高。

接下来我们介绍卷积神经网络（CNN）在自然语言的典型应用。

在自然语言领域，卷积的作用在于利用文字的局部特征。一个词的前后几个词必然和这个词本身相关，这组成该词所代表的词群。词群进而会对段落文字的意思进行影响，决定这个段落到底是正向的还是负向的。对比传统方法，利用词包（Bag of Words），和TF-IDF等，其思想有相通之处。但最大的不同点在于，传统方法是人为构造用于分类的特征，而深度学习中的卷积让神经网络去构造特征。

以上便是卷积在自然语言处理中有着广泛应用的原因。

接下来介绍如何利用Keras搭建卷积神经网络来处理情感分析的分类问题。下面的代码构造了卷积神经网络的结构。

```
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv1D, MaxPooling1D
model = Sequential()
model.add(Embedding(vocab_size, 64, input_length = maxword))
model.add(Conv1D(filters = 64, kernel_size = 3, padding = 'same', activation
 = 'relu'))
model.add(MaxPooling1D(pool_size = 2))
model.add(Dropout(0.25))
model.add(Conv1D(filters = 128, kernel_size = 3, padding = 'same',activation
 = 'relu'))
```

```
model.add(MaxPooling1D(pool_size = 2))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(64, activation = 'relu'))
model.add(Dense(32, activation = 'relu'))
model.add(Dense(1, activation = 'sigmoid'))
model.compile(loss = 'binary_crossentropy', optimizer = 'rmsprop', metrics =
 ['accuracy'])
print(model.summary())
```

下面对模型进行拟合。

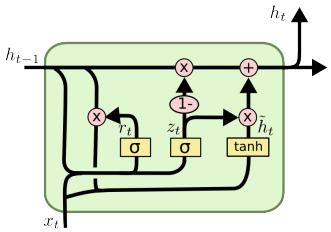
```
model.fit(X_train, y_train, validation_data = (X_test, y_test), epochs = 20,
 batch_size = 100)
scores = model.evaluate(X_test, y_test, verbose = 1)
print(scores)
```

精确度提高了一点，在 85.5% 左右。读者可以试着调整模型的参数，增加训练次数等，或者使用其他的优化方法。这里还要提一句，代码里用了一个 Dropout 的技巧，大致意思是在每个批量训练过程中，对每个节点，不论是在输入层还是隐藏层，都有独立的概率让节点变成 0。这样的好处在于，每次批量训练相当于在不同的小神经网络中进行计算，当训练数据大的时候，每个节点的权重都会被调整过多次。另外，在每次训练的时候，系统会努力在有限的节点和小神经网络中找到最佳的权重，这样可以最大化地找到重要特征，避免过度拟合。这就是为什么 Dropout 会得到广泛的应用。

7.2.4 循环神经网络训练情感分析

下面介绍如何用长短记忆模型（LSTM）处理情感分类。

LSTM 是循环神经网络的一种。本质上，它按照时间顺序，把信息进行有效的整合和筛选，有的信息得到保留，有的信息被丢弃。在时间 t，你获得到的信息（比如对段落文字的理解）理所应当会包含之前的信息（之前提到的事件、人物等）。LSTM 说，根据我手里的训练数据，我得找出一个方法来如何进行有效的信息取舍，从而把最有价值的信息保留到最后。那么最自然的想法是总结出一个规律用来处理前一时刻的信息。由于递归性，在处理前一个时刻信息时，会考虑到再之前的信息，所以到时间 t 时，所有从时间点 1 到现在的信息都或多或少地被保留一部分，也会被丢弃一部分。LSTM 对信息的处理主要通过矩阵的乘积运算来实现的（见图7.3）。



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

图 7.3 长短记忆神经网络示意图（图片来源：http://colah.github.io/posts/2015-08-Understanding-LSTMs/）

构造 LSTM 神经网络的结构可以使用如下的代码。

```
from keras.layers import LSTM
model = Sequential()
model.add(Embedding(vocab_size, 64, input_length = maxword))
model.add(LSTM(128, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(64, return_sequences=True))
model.add(Dropout(0.2))
model.add(LSTM(32))
model.add(Dropout(0.2))
model.add(Dense(1, activation = 'sigmoid'))
```

然后把模型打包。

```
model.compile(loss = 'binary_crossentropy', optimizer = 'rmsprop', metrics =
 ['accuracy'])
print(model.summary())
```

最后输入数据集训练模型。

```
model.fit(X_train, y_train, validation_data = (X_test, y_test), epochs = 5,
batch_size = 100)
scores = model.evaluate(X_test, y_test)
print(scores)
```

预测的精确度大致为 86.7%，读者可以试着调试不同参数和增加循环次数，从而得到更好的效果。

7.3 总结

本章介绍了不同种类的神经网络，有多层神经网络（MLP），卷积神经网络（CNN）和长短记忆模型（LSTM）。它们的共同点是有很多参数，需要通过后向传播来更新参数。CNN 和 LSTM 作为神经网络的不同类型的模型，需要的参数相对较少，这也反映了它们的一个共性：参数共享。这和传统的机器学习原理很类似：对参数或者模型加的限制越多，模型的自由度越小，越不容易过度拟合。反过来，模型参数越多，模型越灵活，越容易拟合噪声，从而对预测造成负面影响。通常，我们通过交叉验证技术选取最优参数（比如，几层模型、每层节点数、Dropout 概率等）。最后需要说明的是，情感分析本质是一个分类问题，是监督学习的一种。除了上述模型，读者也可以试试其他经典机器学习模型，比如 SVM、随机森林、逻辑回归等，并和神经网络模型进行比较。

8 文字生成

8.1 文字生成和聊天机器人

文字信息是存在最广泛的信息形式之一，而深度学习的序列模型（Sequential Model）在对文字生成建模（Generative Model）方面具备独特的优势。文字自动生成可应用于自然语言对话建模和自动文稿生成，极大地提高了零售客服、网络导购以及新闻业的生产效率。目前比较成熟的是单轮对话系统以及基于单轮对话系统的简单多轮对话系统，这类系统应用范围广泛，技术相对成熟，在零售客服、网络导购等领域都有很高的边际收益。例如苹果手机中的 Siri，以及微软早期的小冰机器人，都属于这种系统的尝试。

从应用范围来看，自然对话系统分为所谓的闲聊（Chit-Chat）对话系统和专业（Domain Specific）对话系统两种。

闲聊对话系统的典型代表有苹果的 Siri、微软的小冰机器人等，其构筑于数量极大，超多样化的对话数据上，比如微信群的聊天数据、微博的对话数据等，其特点是应对话题广泛但是不深入。这种对话系统可以应对各种问题或者话题，比如“今天天气好吗？”、“你觉得奇瑞汽车如何？”、“回锅肉好好吃啊”等。这类系统根据建模数据的分布，会分别回应“天气不错”、“其实吉列的博瑞也不错”、“是啊”等。具体的回应是根据系统的建模数据分布以及反馈系统的熵值设定（即多样化设定）。

而专业对话系统主要应用于各种具体的业务领域，比如某类产品的导购或者客服等，建模数据一般都是本领域清洗过的数据。这类系统可以应对的话题比较窄，但是非常深入，有较大可能性是一个多轮对话系统；而对于不涵盖的话题应对方式可以比较简单，因为客户对于比较少见的问题的回答期望较低，而且可以进一步接驳人工服务。

同样对于上面的“今天天气好吗”这样的问题，专业对话系统可以进行更为深入的回应，比如：“今天天气不错，多云间晴，最高气温为 27 摄氏度，最低气温为 21 摄氏度，湿度为 75%”等。在闲聊系统中也可以进行适应性改造，需要进行话题识别，在一定的范围内可以将其增强为专业系统，但是需要做大量的工作和相应的数据。

从技术上看，短对话聊天系统分为两种：基于检索的系统（Retrieval Based System）和基于文字生成的系统（Generative System）。

基于检索的系统鲁棒性强，技术复杂度稍低，但是应对能力有限，比较适合非常窄的专业服务领域。基于检索的对话系统一般基于已有的语义分析理论，对于常见句式进行预先标注，将可能的文字输入分解为不同的部分，然后将相应的部分，比如主语、宾语等，设置为建模对象，实现对具有相似语法结构的不同话题的应对。比较有名的基于检索的对话系统有爱丽丝聊天机器人（Alicebot）及其各种变形。这种聊天机器人基于人工智能标志语言（AIML），通过将现有数据库中不具备的句式载入数据库，具备初级的学习能力，但是它基于一个强假设，即与其聊天的人的回应是正确的，这种假设条件在实际应用中很难满足。

基于文字生成的系统由数据驱动，应对能力强，反馈多样化，有自我学习能力，显得更为智能，但是其技术要求更高，而且系统的鲁棒性较差，对于很多边界条件需要提前考虑到。比如微软研究院于 2016 年推出的 Tay 聊天机器人就对边界条件考虑不周，在上线以后，接收的即时训练数据都是负面的或者不适宜的，但是这个系统在设计时没有考虑到这个问题，仍旧使用这些数据进行训练，造成对话系统的回应也变得很负面，大量用户反馈其爆粗口或者回应不适宜的话题。从技术上解决这个问题并不难，但是要从一开始就考虑到各种各样的这类边界情况就很难了。

8.2 基于检索的对话系统

本节介绍如何利用人工智能标识语言（AIML，Artificial Intelligence Markup Language）搭建基于检索的对话系统。对于比较窄的应用领域，该系统搭建快速、鲁棒性强、具备一定的灵活性，可以和基于文字生成的对话系统组成混合系统，各自取长补短。

AIML 系统具备以下几个优点。

- AIML 系统是基于检索的对话系统中应用比较广泛的一个系统，其不同的变种已经商业化并应用在一些行业实践中。
- 在基于深度学习、数据驱动的对话引擎大量出现之前，AIML 系统基本是基于检索的对话系统中最好的系统之一。

- AIML 系统是开源软件，有不同的语言接口，比如 C++、Java、.NET、Python 等，因此虽然原系统是基于英文的，但是可以很容易地改造为一个中文系统。
- 开源的 AIML 系统已经提供了大量的语法规则供开发者使用和参考，因此开发者可以根据自己的业务场景进行改造，以适应自己的需求。
- AIML 系统是一个引擎，开发者可以很容易地将其嵌入自己的系统中。

AIML 是一种兼容 XML 语法的通用标识语言，非常容易学习。AIML 由一系列包括在尖括号中的元素（Tag）组成，其中一些重要的元素有：

- <aiml>：该元素标识 AIML 文档的起始。
- <category>: 这个元素标识一个知识类，是 AIML 的基本对话构造部件。关于知识类，在后面会详细介绍。
- <pattern>: 该元素标识一个语法模式，是对句式的概括和抽象。
- <template>: 模板元素包含对上面语法模式的回应。
- <that>: 指代对象标识元素。
- <topic>: 上下文归类元素。
- <random> : 随机选择元素，从模板包含的多个回应中随机选择一个，让聊天机器人看起来更智能。
- <srai> : 递归标识元素。
- <think> : 类似于条件控制语句，有针对性地控制回应。
- <get> : 获取 AIML 变量中存储的值。
- <set> : 将一个值存到指定的 AIML 变量中。
- <star> : 通配指代元素。在 <pattern> 元素中可以使用通配符 * 替换任何语言要素，<star> 可以在后面指代这个通配符对应的语言要素。

AIML 的基本知识检索单元被称为一个知识类，用 <category> 元素标注。这个单元包含 3 个子元素：输入模式、应对模板和其他可选项。

输入模式用 <pattern> 元素标识指代，在 AIML 本来的设计中一般是一个问题，但是并不局限于此，其实可以是任意句式。

应对模板使用 <template> 元素标识指代，一般是一个对应的回答或者跟输入的句式相对应的回应句式。比如输入不是一个问题而是一句问候：“你好呀！”回应就可以设定为：“我很好，你好。”或者“最近很忙，你呢?”等。

其他可选项分别是 <that> 对象指代元素和 <topic> 话题指代元素。

使用 AIML 构造基于检索的对话系统非常简单，下面是最简单的一个 AIML 语法库，其中定义了一个标准知识类 <category> 模板：

```
<aiml version="1.0.1" encoding = "UTF-8">
  <category>
     <pattern>你好呀</pattern>
     <template>
              你好，最近很忙呢
     </template>
  </category>
</aiml>
```

在这个模板中定义了一个最简单的知识类，输入句式是：“你好呀”，AIML 会进行精准匹配。回应模板中设计了一个固定的回应模式：“你好，最近很忙呢”。这个知识类就是一个固定的问答模式，只能用于展示 AIML 语法构成。

下面可以进行一些拓展。假设我们想让系统的回应具有一定的多样性，则可以通过设定随机选择元素 <random> 来拓展回应模板。下面的例子中建立了一个随机选择元素，包含 3 种回应方式，AIML 每次遇到能匹配的输入句式时会随机选择一个回应。

```
<aiml version="1.0.1" encoding = "UTF-8">
  <category>
     <pattern>你好呀</pattern>
     <template>
         <random>
              <li>你好，最近很忙呢</li>
              <li>过得不错，你怎么样？</li>
              <li>刚吃了饭，你吃了吗？</li>
         </random>
     </template>
  </category>
</aiml>
```

虽然随机性能让系统的回应更为人性化，但是仍然非常死板。首先，匹配模式不具有灵活性，问候的方式多种多样，我们可能需要一种比较灵活的指定句式的方法。其次，这个回应不能引申到上下文中。

下面使用通配符来改进特定句式的灵活性。对于问候，某些基本句式可以泛化。比如：“你的身体怎么样？”“你的汽车性能怎么样？”等。这时可以用通配符“*”代替

“身体”和“车”等对象，然后在回应中可以使用 <star> 来指代通配符指定的对象，而且可以通过 index= 来指定对应第几个通配符：

```
<aiml version="1.0.1" encoding = "UTF-8">
  <category>
     <pattern>你*怎么样？ </pattern>
     <template>
             我<star/>还好。
     </template>
    </category>

    <category>
     <pattern>*的*好不好？ </pattern>
     <template>
             <star index="1"/>的<star index="2"/>t挺不错的。
     </template>
  </category>
</aiml>
```

多轮对话比单轮对话增加的难度在于对上下文的感知。AIML 提供了两个帮助联系上下文的元素——<that> 和 <topic>。<that> 元素对于前面提到的回应进行进一步的应对。比如有一段对话是：

人：“你喜欢什么？”

程序：“你想聊聊跑车吗？”

这个时候人的可能回答是：喜欢或者不喜欢。根据人的回答，程序应该有不同的回应，这时候就轮到 <that> 元素派上用场了。下面的程序展示了如何应用 <that> 元素构造第二轮回应：

```
<aiml version="1.0.1" encoding = "UTF-8">
  <category>
     <pattern>你喜欢什么？ </pattern>
     <template>
             想聊聊跑车吗？
     </template>
   </category>

   <category>
```

```
        <pattern>可以</pattern>
        <that>想聊聊跑车吗？</that>
        <template>
             <random>
                 <li> 太好了，我喜欢美式大排量跑车。 </li>
                 <li> 太好了，我喜欢小排量高转引擎跑车。 </li>
                 <li>真棒，我就喜欢贵的跑车。</li>
             </random>
        </template>
    </category>

    <category>
        <pattern>行啊</pattern>
        <that>想聊聊跑车吗？</that>
        <template>
             <random>
                 <li> 太好了，我喜欢美式大排量跑车。 </li>
                 <li> 太好了，我喜欢小排量高转引擎跑车。 </li>
                 <li>真棒，我就喜欢贵的跑车。</li>
             </random>
        </template>
    </category>

    <category>
        <pattern>想啊</pattern>
        <that>想聊聊跑车吗？</that>
        <template>
             <random>
                 <li> 太好了，我喜欢美式大排量跑车。 </li>
                 <li> 太好了，我喜欢小排量高转引擎跑车。 </li>
                 <li>真棒，我就喜欢贵的跑车。</li>
             </random>
        </template>
    </category>

    <category>
        <pattern>不想</pattern>
```

```
        <that>想聊聊跑车吗？ </that>
        <template>
             <random>
                 <li> 哎，那聊电影怎么样？ </li>
                 <li> 不喜欢车啊，太无趣了。 </li>
             </random>
        </template>
    </category>
</aiml>
```

在这个例子里，我们针对“想聊聊跑车吗?”这个第一轮回应，做出了 4 种不同的第二轮回应，其中有 3 种是对应肯定回答的回应，一个是对应否定回答的回应，在每一个回应中，在 <pattern> 元素后面使用 <that> 元素来引用第一轮的回应。

<topic> 元素用来定义一段对话的知识类以供以后检索对应回答。类似于 <that> 元素，该元素也通常使用于肯定与否定的回答中，不同的是，该元素保存的是整个知识类，而不是某一个具体回应。比如下面的对话就可以用 <topic> 元素来帮助回应：

人：“让我们聊聊跑车吧。”

程序：“好的，聊聊跑车。”（这时候定义跑车这个 topic。）

人：“美式跑车就不错。”

程序：“一说到跑车就兴奋。”

人：“我特别喜欢美式大排量引擎的跑车。”

程序：“我也喜欢美式大排量引擎跑车。”

这段对话可以使用下面的 AIML 程序来设置：

```
<aiml version="1.0.1" encoding = "UTF-8">
  <category>
     <pattern>让我们聊聊跑车吧</pattern>
     <template>
             好的，聊聊 <set name="topic">跑车</set>。
     </template>
  </category>

  <topic name="跑车">
     <category>
        <pattern> * </pattern>
```

```
            <template>
                  一说到跑车就兴奋
            </template>
      </category>

      <category>
            <pattern>我特别喜欢*跑车</pattern>
            <template>
                        我也喜欢<star/>跑车
            </template>
      </category>

   </topic>
</aiml>
```

我们可以看到，使用 <topic> 元素可以构造比较灵活的上下文敏感的简单多轮对话系统。

但是我们也看到，在应用 <that> 元素的例子里，针对 3 个正面回应，第二轮的回应都是一样的，但是需要写 3 个知识类，非常烦琐，有没有什么办法简化呢？这时候就需要用到递归元素 <srai>。该元素让 AIML 能够对同一个模板定义不同的目标，从而不仅能简化同类型回应，还有其他非常强大的作用，使机器人的回应显得更加拟人化。

递归元素能用于解决以下几类问题：

- 句式归一（Symbolic Reduction）
- 分治（Divide and Conquer）
- 同义词解析（Synonyms Resolution）
- 关键词检测（Keyword Detection）

句式归一是用来简化句式的一种方法，其能将复杂的句式分解为较为简单的句式，反过来说，就是用以前定义的较为简单的句式来重新定义复杂的句式。举例来讲，一个问题可以有多种提法，比如，我们可以问：“谁是黄晓明？”也可以这么问：“你认识黄晓明吗？”而且“黄晓明”这个名字也可以替换成任何人。因为这是一个类型语义的多种不同表达方式，都有一定的句式：要么以“谁是”开头，要么以“你认识”开头，因此可以使用 <srai> 进行句式归一。我们先对第一个问题的句式建立一个知识类：

```
<category>
   <pattern>谁是黄晓明？ </pattern>
```

```
    <template>黄晓明是一名中国演员</template>
</category>

<category>
    <pattern>谁是马化腾？</pattern>
    <template>马化腾是一名中国企业家，QQ软件的发明者，董事会主席兼首席执行官
    </template>
</category>
```

然后可以通过 <srai> 引申为一个更为通用的句式范畴，并和其他同样语义的句式归一：

```
<category>
    <pattern>你认识*吗?</pattern>
    <template>
        <srai>谁是 <star/></srai>
    </template>
</category>
```

首先将询问的对象用通配符“*”泛化，然后再通过 <srai> 把类似的问句归纳到第一个知识类的句式，这样系统就能自动匹配并应对多样化的问句。

分治功能主要是通过重用一个知识类句式的一部分来减少重复定义的句式。比如说到再见，可以说“再见了”；或者“再见，哥们儿”；或者“再见，某某某”等。只要以“再见”开头的句式一般都是道别，因此对于类似的道别句式，都可以归纳到一个回应模板中。下面的例子就展示这个功能：

```
    <category>
        <pattern>再见</pattern>
        <template>再见啦！</template>
    </category>

    <category>
        <pattern>再见*</pattern>
        <template>
            <srai>再见</srai>
        </template>
    </category>
```

同义词解析这个功能很直白，即对于具有同样含义的对象，应该有同样的理解。这一点与上面的句式归一功能很类似，用法也几乎一样，只是其不是定义在句式上，而是定义在句子里的一个对象上。

关键字检测是指当在输入的句子中包含某一个特定对象时，AIML 会有一个标准的回应。比如，如果句子中提到“帕拉梅拉”，则 AIML 就会回应：“帕拉梅拉是保时捷产四座豪华轿跑，操控极佳，我喜欢”。下面的代码就实现这个功能：

```
<category>
    <pattern>帕拉梅拉</pattern>
    <template>帕拉梅拉是保时捷产四座豪华轿跑，操控极佳，我喜欢。</template
    >
 </category>

 <category>
    <pattern>_ 帕拉梅拉</pattern>
    <template>
       <srai>帕拉梅拉</srai>
    </template>
 </category>

 <category>
    <pattern>帕拉梅拉 *</pattern>
    <template>
       <srai>帕拉梅拉</srai>
    </template>
 </category>

 <category>
    <pattern>_ 帕拉梅拉 *</pattern>
    <template>
       <srai>帕拉梅拉</srai>
    </template>
 </category>
```

这里使用了前缀通配符“_”和一般通配符“*”来匹配任意句式。

前面提到过，AIML 是一个检索式对话引擎，可以嵌入任何系统中提供服务。下面尝试在 Jupyter Notebook 里面构造一个超简易问答。先给 Notebook 定义一个宏变量

——%%ask，让 Notebook 知道后面的输入信息应该返回给 AIML 引擎进行处理并返回相应的信息。这样就可以在 Jupyter Notebook 里面进行问答了，如图8.1所示。

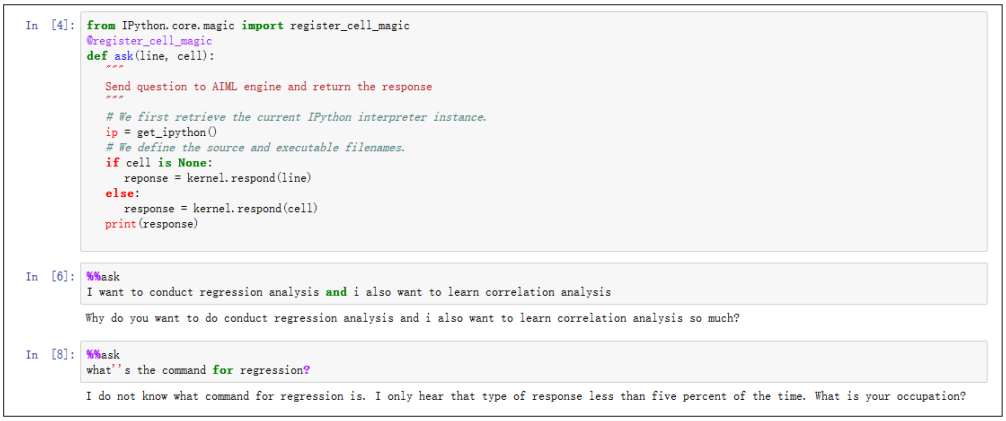


图 8.1 Jupyter Notebook 里通过宏命令向 AIML 提问

首先定义自己的宏变量：

```
from IPython.core.magic import register_cell_magic
@register_cell_magic
def ask(line, cell):
    """
    Send question to AIML engine and return the response
    """
    ip = get_ipython()
    if cell is None:
        reponse = kernel.respond(line)
    else:
        response = kernel.respond(cell)
    print(response)
```

其中，ip=get_ipython() 命令获取当前 IPython 环境对象，紧随其后的判断语句定义信息来源（cell / line）和执行内核（kernel）。最后打印输出。

然后需要载入 AIML 引擎，定义内核和相应的数据库：

```
import aiml
kernel = aiml.Kernel()
kernel.learn("d:\\data\\project\\aiml\\std-startup.xml")
kernel.respond("LOAD BRAIN")
```

第一行命令是载入引擎，第二行命令是定义内核，第三行命令是定义数据库的调用脚本，最后的 LOAD BRAIN 载入数据库。这个标准调用脚本 XML 文件很简单，就是定义了到什么地方搜索 *.aiml 数据库文件，这些数据库文件就是前面例子里的 AIML 文本文件。这个 XML 脚本内容如下：

```
<aiml version="1.0.1" encoding="UTF-8">
    <!-- std-startup.xml -->
    <!-- Category is an atomic AIML unit -->
    <category>

        <!-- Pattern to match in user input -->
        <!-- If user enters "LOAD AIML B" -->
        <pattern>LOAD BRAIN</pattern>

        <!-- Template is the response to the pattern -->
        <!-- This learn an aiml file -->
        <template>
            <learn>d:\\data\\project\\aiml\\standard\\*.aiml</learn>
            <!-- You can add more aiml files here -->
            <!--<learn>more_aiml.aiml</learn>-->
        </template>
    </category>
</aiml>
```

其非常类似于普通的 AIML 数据库文本文件，都是用 <category> 定义一个知识点，这里的句式是固定的 LOAD BRIAN，回应就是具体的调用操作 <learn>：

```
<learn>d:\\data\\project\\aiml\\standard\\*.aiml</learn>
```

因此才能在最后一个命令中要求对 LOAD BRAIN 输入做出动作就产生载入数据库的行为。

调入相应的库以后，就可以用刚刚定义好的%ask 宏命令在 Jupyter Notebook 中提问了。

因为在我们定义的问题库里面没有提问的这些句式和对象，所以 AIML 无法很好地回应。

当然，利用平时收集的业务数据，读者可以根据自己的业务需求快速搭建一些有针对性的简易应答机器人，对于常见问题，它们是能够较好地回答的。如果要搭建更为智能的对话机器人，则需要继续学习下面两节的内容。

8.3 基于深度学习的检索式对话系统

上面的基于 AIML 的聊天对话系统虽然已经成功应用于早期的商业客服领域，但是有几个大的问题阻碍了其更大范围的应用。

首先，建立对话库需要大量的人力和时间的积累。虽然 AIML 系统开源项目已经公开了数十万句英语对话，但是要将其转换为中文，仍然需要做大量的工作，而且这些对话仍然基于一般的聊天场景，并不针对某一个具体业务，而这类系统最实用的应用是范围比较窄的具体业务，如果要建立这套应用系统，则工作量仍然是巨大的。

其次，虽然 AIML 系统提供了一些功能赋予了聊天对话系统一定的灵活性，比如对对象的记忆、通配、递归等，但是其功能仍然受到很大限制。

再次，AIML 系统在本质上仍然是一个检索性质的系统，而且属于硬性匹配，对于没有见过的询问模式，会无法检索到答案，影响用户体验。

基于深度学习的检索式对话系统在后两点上有较大的改进。通过机器学习，可以实现软匹配，不需要询问语句模式一定要在库里出现，如果通过对词汇和其组织顺序的建模发现有较高概率匹配的数据点，就能实现比较好的回应。其次，深度学习方法提供了更高的灵活性，能够实现记忆和识别等功能。下面通过一个简单的例子来学习如果在 Keras 里训练基于深度学习的检索式对话系统。

8.3.1 对话数据的构造

在讨论建模之前，一个很重要的工作是对数据进行处理。公开的对话数据，特别是带标定的可以用于索引式对话模型建模的中文对话数据不是很好找。这里使用的训练数据是英文的 Ubuntu 论坛讨论数据，该数据是目前较大的带标定的对话数据，由 McGill 大学的 Ryan Lowe、Nissan Pow、Iulian V. Serban 和 Joelle Pineau 创建。他们根据这个数据构造了一个索引式深度学习多轮对话系统，发表于 SIGDial 2015 会议上。

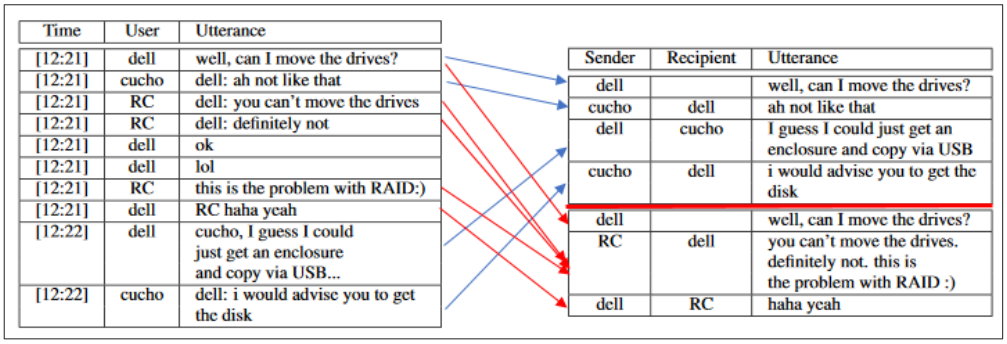
读者可以去以下网站下载原始数据：http://dataset.cs.mcgill.ca/ubuntu-corpus-1.0/。

我们也为读者下载好了包括原始数据和处理以后的二进制 pickle 数据放在本书的下载资料中，方便读者下载。

下面要讨论一下这个数据的构造，这样读者在将其应用到自己的项目时就知道如何从原始对话数据开始对其进行组织和标定，从而能够纳入下面的建模框架进行建模。Ubuntu 对话数据的特点非常适合特定领域自动客服和导购对话系统的构建，因为这类对话具有以下特定的性质。

- 对话内容涵盖领域非常具体，涉及内容比较窄。
- 双人应答式多轮对话。
- 数据量大，一般要求数百万条对话以供训练深度学习模型。

图8.2展示了从一个原始对话扩展为可建模用双人多轮对话的过程。



Time	User	Utterance
[12:21]	dell	well, can I move the drives?
[12:21]	cucho	dell: ah not like that
[12:21]	RC	dell: you can't move the drives
[12:21]	RC	dell: definitely not
[12:21]	dell	ok
[12:21]	dell	lol
[12:21]	RC	this is the problem with RAID:)
[12:21]	dell	RC haha yeah
[12:22]	dell	cucho, I guess I could just get an enclosure and copy via USB...
[12:22]	cucho	dell: i would advise you to get the disk

Sender	Recipient	Utterance
dell		well, can I move the drives?
cucho	dell	ah not like that
dell	cucho	I guess I could just get an enclosure and copy via USB
cucho	dell	i would advise you to get the disk
dell		well, can I move the drives?
RC	dell	you can't move the drives. definitely not. this is the problem with RAID :)
dell	RC	haha yeah

图 8.2　Ubuntu 多人多轮对话处理展示

从图8.2可以看出，一个长对话根据标识被划分为两人之间的对话，如果回答没有引用是针对谁，比如图8.2中的用户 RC 的回答，那么这两次没有应用客户的回答都被归入上一次最近引用的用户的回应中。

这里要强调的是，很多读者可能认为在自己的内部系统中，客服数据已经是一个很干净的单一话题双人多轮对话数据，为何这里还要介绍如何处理这样一个不是很像内部系统数据的数据处理过程？这是因为在实际业务中，可能某一个单次对话是典型的双人多轮对话，但是很多时候一次客户服务的过程并不能解决用户的疑问，用户有时会间隔一两天或者更长时间再跟客服联系，要求解决一样的问题，这在一些比较复杂的业务场景中是经常出现的。比如作者以前所在的财险公司，当保险客户需要增加保险标的，或者修改保险条款时，有超过 25% 的情况需要联系客服一次以上，因为这些业务问题往往涉及多个方面，比如修改保险标的会涉及多标的折扣、条款修改，新文档寄送，以及增值服务销售等，有时候客服或者用户不能一次想到所有的相关问题。如果仍然只用一次联系的数据作为训练，那么训练结果不能反映这种深刻的联系，但是如果能解决这种相关话题的涵盖问题，不仅能提高客服对话系统的智能度，还能用于训练新的人工客服。

因为这里是要建立一个索引式的深度学习对话系统，所以在本质上我们需要建立一个基于深度学习算法并且还有记忆功能的分类模型，那么我们还需要对这组已经分成双人多轮对话的数据进行进一步处理，具体包括以下三个部分。

(1) 首先，需要将整个对话分成两个部分，即背景 + 回应。背景是一段对话在截止回应之前的所有相关文字，如果有多轮对话，则各段文字用一个特定符号分开。回应就是截止该轮对话之后的立即回应。比如针对前面那个叫 dell 的用户和 cucho 的用户的对话，在第二轮对话之后，其背景是两人对话的前两句：

dell : well, can I move the drives?

cucho: ah not like that.

而回应则是 dell 对 cucho 说的第 2 句话：

I guess I could just get an enclosure and copy via USB.

如果是在第三轮对话之后，其背景是对话的前三句：

dell : well, can I move the drives?

cucho: ah not like that.

dell: I guess I could just get an enclosure and copy via USB.

而回应则是 cucho 对 dell 说的第二句话：

I would advise you to get the disk.

注意，这里背景的文字是将所有本轮对话之前的对话集合起来，用特殊字符串分开。比如原作者会将上面的背景写成：

well, can I move the drives _EOS_ ah not like that _EOS_ I guess I could just get an enclosure and copy via USB.

这里不使用 “;” 或者 “,” “。” 等标点符号是因为原有语句的标点可能包含了这些常用或者不常用的标点符号，因此非常特殊的专门构造的分割字符串是比较理想的选择。

在这里用户 ID 不需要出现。

(2) 其次，因为这是一个分类问题，因此需要生成正确的回应和一个或者多个不正确的回应，这样深度学习算法才能训练模型。读者可能要问，在标准的对话系统里，回应肯定都是针对问题的，都是正确的，怎么找出不正确的回应呢？这里是通过在与此次对话不相关的其他对话的回应中进行随机取样获得。根据具体情况，可以随机取样一个不相关的回应，或者多个不相关的回应。

(3) 最后，对正确的回应及其背景，生成一个标识为 1，属于正样本；对于同样的背景和随机抽取的不正确回应配对，生成一个标识为 0 或者 −1，属于负样本。

如此处理以后的数据就是一个三维对话数，如表8.1所示。

表 8.1　最终数据集的构造

背景	回应	标识
well, can I move the drives _EOS_ ah not like that	I guess I could just get an enclosure and copy via USB	1
well, can I move the drives _EOS_ ah not like that	That's interesting	0
well, can I move the drives _EOS_ ah not like that	Prior to applying the method you need to fix something	0
well, can I move the drives _EOS_ ah not like that _EOS_ I guess I could just get an enclosure and copy via USB	I would advise you to get the disk	1
well, can I move the drives _EOS_ ah not like that _EOS_ I guess I could just get an enclosure and copy via USB	lol	0
...	...	...

8.3.2　构造深度学习索引模型

完成数据处理以后，要运用上面的三维对话数据构造索引式对话系统，则可以使用一个基于深度学习的分类模型来进行。一般来说，一个分类模型建模包含以下几部分：

- 选择模型。
- 数据预处理使其适合所用模型。
- 对模型进行拟合。
- 检验模型性能，结合上一步调参。

这里我们选用原作者的双编码长短记忆（Dual Encoder LSTM）模型。该模型在原文中有最好的性能，因为 LSTM 能够记忆较长的内容，比较适合多轮对话的场景。这个模型的结构如图8.3所示。

图8.3展示了对一段对话的背景和回应分别编码建立长短记忆模型，再合并计算余弦相似度的结构。图8.3中的上半部分循环时间模型对应的是背景部分的数据，其中 c_t 对应 t 时刻的背景信息，h_t 则是状态变量，下半部分对应的是应答模型，其中 r_t 是 t 时刻的回应。函数 σ 则是合并函数。

对于含有文本的数据，正如在第 1 章提到的，一般先要进行必要的预处理使其变为索引数字。对于英文文本，这些预处理包括 tokenization、stemming、lemmantization 等。对于中文文本，这些预处理包含分词等操作。这些操作结束以后，经过处理后的每一个单词或者单字都用来建立一个索引，并且被分配一个索引下标号，这样对文本的建模就变为对索引标号这种整数的建模。

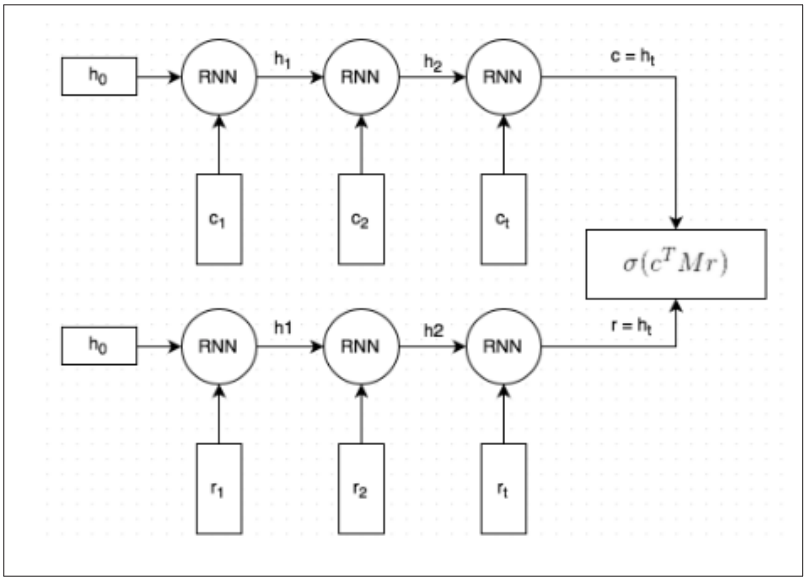


图 8.3 双编码长短记忆模型

得到了索引标号以后，每一段对话的背景和回应都是一组整数。最长的一段对话的背景有 2002 个索引号，最短的有 3 个索引号，平均有 162.5 个索引号，中位数则是 120 个索引号，因此这个数据的维度还是很高的。我们可以进行几个进一的操作来帮助建模。

首先，最自然的方法是做向量嵌入（Embedding）操作，将高维的索引标号数据投影到一个固定低维度的实数向量中。其次，因为每段对话长短不一，需要将其长度标准化，即补齐工作。对于补齐工作，并不是将所有向量补齐为最长的 2002 个索引号，而是选择一个合理的长度，通过移动窗口将上下文纳入不同时间步构造一个不是太宽的叠加起来的向量。这里可以选择中位数 120 作为补齐长度。

对于一组索引下标向量，用 Keras 做向量嵌入最简单的方法就是将每个长短不一的向量补齐以后用 Embedding 方法进行映射，以上操作可以通过下面的代码实现。

```
def batch_generator(X, max_length=20, batch_size=1):
    number_of_batches = samples_per_epoch/batch_size
    counter=0
    shuffle_index = np.arange(len(X))
    np.random.shuffle(shuffle_index)
    X =  X[shuffle_index, :]
    while 1:
```

```
        index_batch = shuffle_index[batch_size*counter:batch_size*(counter
        +1)]
        Xtemp = [].extend(X[index_batch])
        status = np.sum([isinstance(e, list) for e in Xtemp])
        if (status==0):
            Xtemp = [Xtemp]
        X_batch = pad_sequences(Xtemp, maxlen=max_length)
        X_res = np.zeros((np.shape(X_batch)[0], maxlen, max_length), dtype=
        np.int)
        for k in np.range(np.shape(X_batch)[0]):
            X_res[k, 0, :] = X_batch[k, :]
        counter += 1
        yield(X_res)
        if (counter < number_of_batches):
            np.random.shuffle(shuffle_index)
            counter=0
```

这里首先使用了 data generator 函数，针对一组批量的输入索引下标向量列表进行补齐，否则一次性对所有样本数据进行操作内存消耗太大。在这个 data generator 里，先从原始数据里抽取一组指定批量数大小的样本，这组样本是下标标识向量列表，而这些下标标识对应的就是不定长的句子里面每个字的索引号。对于这组列表，按照指定长度进行向量嵌入有两种方法：一种是指定长度大于或等于最长句子的长度，这样每段对话都可以用一个向量表达，能够对所有列表中的向量一次进行补齐；第二种是指定长度短于最长句子的长度，比如用中位值作为指定长度，这时候对于不够指定长度的较短向量可以直接补齐，而对于超过指定长度的向量，则按照指定的长度建立移动窗口，建立不同时间步对应的对话上下文向量列表，最后对补齐的输出进行映射。不论是哪种方法，都需要生成三维的张量，中间维度是时间步，对于第一种情况，时间步长度永远为 1，对于第二种情况，时间步长度则是可变的。

另外，这里的 status = np.sum([isinstance(e, list) for e in Xtemp]) 语句用来判断返回的样本是否是列表的列表，因为 pad_sequences 只能对列表的列表进行操作，否则会出错。

最后两个命令分别为背景对话和应答定义了通用模型，并添加一个向量嵌入层作为首层网络。这个嵌入层将 2002 维的索引向量映射到一个只有 256 维的较低维度的致密实数向量中。这里选择 256 维是考虑到我们的硬件限制。如果硬件条件允许，则也可以将这个维度稍微提高一点。比如先映射到 512 维，再通过几个全连接层逐次降维到 256 维再输入长短记忆层进行处理。

得到映射好的实数向量以后，可以直接添加一个长短记忆层（LSTM），这个层

接受一个对话的相关信息，并输出 128 位的神经元向量。其次，我们需要对背景对话和相应的应答分别建立循环神经网络。在第 1 章提到了，Keras 的循环神经网络本身是一个抽象类，因此我们只能用其 3 个实现的子类来构造具体的循环神经网络，比如 SimpleRNN、LSTM 和 GRU。这里采用了 LSTM，分别对背景对话和应答进行建模。用 LSTM 进行建模在 Keras 里非常容易。首先建立一个序列模型，然后在序列模型中添加相应的 LSTM 网络层即可。

```
encoder_context = Sequential()
encoder_context.add(LSTM(32, input_shape=(timesteps, data_dim)))

encoder_response = Sequential()
encoder_response.add(LSTM(32, input_shape=(timesteps, data_dim)))
```

最后，将两个不同的 LSTM 网络通过运用张量点乘的方法合并，然后在其上训练一个全连接网络。

```
decoder = Sequential()
decoder.add(Merge([encoder_context, encoder_response], mode='dot'))
decoder.add(Dense(32, activation='relu'))
decoder.add(Dense(nb_classes, activation='softmax'))

decoder.compile(loss='categorical_crossentropy', optimizer='rmsprop')
```

最后对于这类对话系统的评估需要一个标准。一般来说，这种模型的评价采用 recall@N 指标。

这个指标指的是，对于一个有 M 个正负例子（一般是 1 个正例子，$M-1$ 个负例子，$M \geqslant 2$）的分类模型，模型根据打分结果选出前 N 个可能的最佳选择（$N \leqslant M$），如果正例子在这 N 个选择里面，那么这个打分选择就标注为正确。如果 $N = M$，因为一共只有 M 个选择，那么 recall@M 一定是 100%。因此最难的指标是 recall@1，因为只有一次选择，最简单的是 recall@M，因为结果总是 100%。

recall@N 的计算通过下面的程序可以方便地获得：

```
def recall(ypred, yactual, N=1):
    num_examples = float(len(y))
    num_correct = 0
    for predictions, actual in zip(ypred, yactual):
        if label in predictions[:N]:
            num_correct += 1
    return num_correct/num_examples
```

8.4 基于文字生成的对话系统

在上面的基于索引信息取回式的对话系统中，如果在一个问句或者对话背景中找不到一个比较好的匹配，那么机器所选择的回应可能会显得风马牛不相及。这时有以下两种选择。

一是当匹配应答的分数低于一个阈值的时候，选择给定一个默认应答，一般表示该系统无法应对，比如“我不理解您的问题，能不能换一种说法？”之类来提示用户。二是使用基于文字生成的对话机器，在更多的情况下尽可能地提供更为智能的应答。

本节会介绍如何使用循环神经网络自动生成智能应答。

这里使用作家老舍的小说《四世同堂》作为训练数据进行演示。读者可以根据自己的应用和业务环境，选择合适的数据。

在很多英文环境的生成式对话系统中，建模的单位有单字和单字符两种。前者给每一个进行过预处理的单词词根建立索引，然后使用单词在该索引上的映射作为原始数据来建模。而后者对于每个英文字符，包括大小写字母、阿拉伯数字以及其他字符等建立索引，然后根据每个单字符在该索引上的映射作为原始数据来建模。在中文里也有相应的两种情况。一种是对进行中文分词以后的词组建立索引并建模，另外一种就是对每个中文单字以及符号等建立索引并建模。

根据以往的经验，在英文环境下，很多基于单字符进行建模都取得了不错的效果，而在中文环境下，中文分词有时候是一个问题，很多具体业务需要建立自己的分词库来准确得到对应的词组，如果采用已有的通用分词库，则很有可能不能得到较好的分词效果，对于一些新出现的词组不能有效进行分割，但是这些新的词组往往是业务发展的体现，因此，即使是较少的错误分词仍然可能造成较大的问题。比如在没有及时更新词库的情况下，“……和美国总统川普通话”这句话会被切分为 [和，美国，总统，

川，普通话]。因此为了简化建模流程，在没有专门制定分词库的情况下，我们选择对单个中文字及相关符号进行建模，这样就跳过了建立自己的分词库或者使用通用分词库但是分词效果不一定好的情况。

对于训练文本可以一次性读入：

```
alltext = open("e:\\data\\Text\\四世同堂.txt", encoding='utf-8').read()
```

得到的结果是一个巨大的字符串列表。因为我们将以每个单字作为建模对象，因此这样读入数据是最方便以后操作的。如果要以词组和单句进行建模，则分段读入最佳。《四世同堂》这本书一共有 3545 个不重复的单字和符号。

按照对文字序列建模的顺序，我们依次进行下面的操作。

(1) 首先对所有待建模的单字和字符进行索引。

(2) 其次构造句子序列。

(3) 然后建立神经网络模型，对索引标号序列进行向量嵌入后的向量构造长短记忆神经网络。

(4) 最后我们来检验建模效果。

对单字和字符建立索引非常简单，用下面三句命令即可：

```
sortedcharset = sorted(set(alltext))
char_indices = dict((c, i) for i, c in enumerate(sortedcharset))
indices_char = dict((i, c) for i, c in enumerate(sortedcharset))
```

第一句的对用 set 函数抽取的每个单字的集合按照编码从小到大排序。第二句对每一个单字进行编号索引，第三句进行反向操作，对每个索引建立单字的词典，主要是为了方便预测出来的索引标号向量转换为人能够阅读的文字。

构造句子序列也非常简单：

```
maxlen = 40
step = 3
sentences = []
next_chars = []
for i in range(0, len(alltext) - maxlen, step):
    sentences.append(alltext[i: i + maxlen])
    next_chars.append(alltext[i + maxlen])
print('nb sequences:', len(sentences))
```

构造句子序列的原因是原始数据是单字列表，因此需要人为构造句子的序列来模仿句子序列。在上面的代码中，maxlen=40 标识人工构造的句子长度为 40 个单字，step=3 表示在构造句子时每次跳过 3 个单字，比如用这一串单字列表“这首小令是李清照的奠定才女地位之作，轰动朝野。传闻就是这首词，使得赵明诚日夜作相思之梦，充分说明了这首小令在当时引起的轰动。又说此词是化用韩偓《懒起》诗意。”来构造句子的时候，假设句子长度为 10，那么第一句是“这首小令是李清照的奠”，而第二句则是移动 3 个单字以后的“令是李清照的奠定才女”。跳字的目的是为了增加句子与句子之间的变化，否则每两个相邻句子之间只有一个单字的差异，但是这两个相邻句子是用来构造前后对话序列的，缺乏变化使得建模效果不好。当然，如果跳字太多，那么会大大降低数据量。比如《四世同堂》一共有 711501 个单字和符号，每隔三个字或者符号进行跳字操作构造的句子只有 237154 个，是原数据量的 1/3。如何选择跳字的个数是读者在建模的时候要根据情况调整的一个参数。

需要注意的是，因为句子是人工构造的，都有固定的长度，因此这里不需要进行句子补齐操作。同时，这些句子的向量其实都是一个稀疏矩阵，因为它们只将包含数据的索引编号计入。

人工构造句子完毕后就可以对其矩阵化，即对于每一句话，将其中的索引标号映射到所有出现的单字和符号，每一句话所对应的 40 个字符的向量被投影到一个 3545 个元素的向量中，在这个向量中，如果某个元素出现在这句话中，则其值为 1，否则为 0。下面的代码执行这个操作：

```
print('Vectorization...')
X = np.zeros((len(sentences), maxlen, len(sortedcharset)), dtype=np.bool)
y = np.zeros((len(sentences), len(sortedcharset)), dtype=np.bool)
for i, sentence in enumerate(sentences):
    if (i % 30000 == 0):
        print(i)
        for t in range(maxlen):
            char=sentence[t]
            X[i, t, char_indices[char]] = 1
    y[i, char_indices[next_chars[i]]] = 1
```

当然这个新生成的数据会非常大，比如 X 会是一个 237154 × 40 × 3545 的实数矩阵，实际计算的时候占用的内存会超过 20GB。因此这里需要使用前面提到的数据生成器（data generator）方法，对一个具有较小批量数的样本进行投影操作。可以通过下面这个很简单的函数实现：

```
def data_generator(X, y, batch_size):
    if batch_size<1:
       batch_size=256
    number_of_batches = X.shape[0]//batch_size
    counter=0
    shuffle_index = np.arange(np.shape(y)[0])
    np.random.shuffle(shuffle_index)
    #reset generator
    while 1:
        index_batch = shuffle_index[batch_size*counter:batch_size*(counter
        +1)]
        X_batch = (X[index_batch,:,:]).astype('float32')
        y_batch = (y[index_batch,:]).astype('float32')
        counter += 1
        yield(np.array(X_batch),y_batch)
        if (counter < number_of_batches):
            np.random.shuffle(shuffle_index)
            counter=0
```

这个函数与前面的 batch_generator 函数非常相似，主要区别是这个函数同时处理 X 和 Y 矩阵的小批量生成，另外要求输入和输出数据都是 NumPy 多维矩阵而不是列表的列表。另外 Python 里的数值数据是 float64 类型的，因此专门使用 astype('float32') 将矩阵的数据类型强制定为 32 位浮点数以符合 CNTK 对数据类型的要求，这样不需要在后台再进行数据类型转换，从而提高效率。现在可以构造我们的长短记忆神经网络模型了。这时候再次体现了 Keras 的高效建模能力。下面短短几个命令就可以让我们构造一个深度学习模型：

```
# build the model: a single LSTM
batch_size=256
print('Build model...')
model = Sequential()
model.add(LSTM(256, batch_size=batch_size, input_shape=(maxlen, len(
sortedcharset)), recurrent_dropout=0.1, dropout=0.1))
model.add(Dense(len(sortedcharset)))
model.add(Activation('softmax'))

optimizer = RMSprop(lr=0.01)
model.compile(loss='categorical_crossentropy', optimizer=optimizer)
```

其中第一句命令指定要生成一个序列模型，第二到第四句命令要求依次添加三层网络，分别是一个长短记忆网络和一个全连接网络，最后使用一个 softmax 的激活层输出预测。在长短记忆网络里，规定输入数据的维度为（时间步数，所有出现的不重复字符的个数），即输入的数据是对应每一句话处理以后的形式，并且对输入神经元权重和隐藏状态权重分别设定了 10% 的放弃率。全连接层的输出维度为所有字符的个数，方便最后的激活函数计算。最后两条命令指定网络优化算法的参数，比如里面指定损失函数为典型的 categorical_crossentropy，优化算法是指定的学习速率为 0.01 的 RMSprop 算法。对于循环神经网络，这个优化算法通常表现较好。

最后我们开始训练模型：

```
model.fit_generator(data_generator(X, y, batch_size), steps_per_epoch=X.
shape[0]//batch_size, epochs=50)
```

这里使用 fit_generator 方法，而不是我们平时所用的 fit 方法，数据输入也是通过 `data_generator()` 函数，fit_generator 将每个批量的数据读入，从稀疏矩阵变为密集矩阵，然后计算。这样对内存的压力大大降低了。下面展示了拟合过程前 5 个迭代的时间和损失函数的值：

```
Epoch 1/50
926/926 [==============================] - 352s - loss: 9.4287
Epoch 2/50
926/926 [==============================] - 352s - loss: 6.3527
Epoch 3/50
926/926 [==============================] - 349s - loss: 6.1262
Epoch 4/50
926/926 [==============================] - 351s - loss: 6.1481
Epoch 5/50
926/926 [==============================] - 350s - loss: 6.1949
```

如果不强制转换数据类型，则运行时间会增加大约 110 秒。最后来看一看效果。先随机抽取一组 40 个连续的字符，然后生成对应的投影到所有字符空间的自变量 x：

```
start_index=2799
sentence = alltext[start_index: start_index + maxlen]
sentence0=sentence
x = np.zeros((1, maxlen, len(sortedcharset)))
for t, char in enumerate(sentence):
      x[0, t, char_indices[char]] = 1.
```

接下来依次对每一句的下 20 个字符进行预测，并根据预测得到的索引标号找出对应的文字供人阅读：

```
generated=''
ntimes = 20
for i in range(ntimes):
    preds = model.predict(x, verbose=0)[0]
    next_index = sample(preds, 0.1)
    next_char = indices_char[next_index]
    generated+=next_char
    sentence = sentence[1:]+next_char
```

读者可能会留意到这里有一个 sample 函数，用于从预测结果得到新生成的文字。这是因为这个模型返回的是对应于每个字符在下一句里的出现概率，而这个函数就是负责根据这个得到的概率对所有索引标号进行依概率的随机取样。但是这个函数还有第二个参数用来控制概率差异的扩大或者缩小。这个参数通常被称为“温度（temperature）”参数。其作用与语句 preds = np.log(preds) / temperature，当温度参数为 1 时，对预测概率没有影响；当温度参数小于 1 时，预测概率的差异被扩大，有利于增加生成语句的多样性；当温度参数大于 1 时，预测概率的差异被缩小，会缩小生成语句的多样性，即在很多时候生成的语句都非常类似，会有很多单字不断地重复出现。通常来说，温度参数应该设置比较小，我们的实验通常设置在小于 0.1 的水平。这个 sample 函数的代码如下：

```
def sample(preds, temperature=1.0):
    preds = np.asarray(preds).astype('float64')
    scaled_preds = preds ** (1/temperature)
    preds = scaled_preds / np.sum(scaled_preds)
    probas = np.random.multinomial(1, preds, 1)
    return np.argmax(probas)
```

那么结果如何呢？我们随机选择的字符串是下面这样的：

一句，小顺儿的妈点一次头，或说一声“是”。老人的话，她已经听过起码有五十次，但是

而生成的字符串是这样的：

不 知 道 ， 他 的 心 中 就 是 一 个 人 ， 而 只 觉 得 自 己

乍一看，其实还读的通，但是仔细读就发现整个句子跟上下文并无任何关系。原因可能是模型不够复杂，无法抓取很多潜在信息；也很可能是数据量太小，一般来说，训练这样一种生成式模型需要数百万条语句才能得到较好的结果，而《四世同堂》这一

部小说是达不到这个水平的。不过在实际应用中，一般公司都会积累大量的客服对话数据，因此这个数据量不会成为模型瓶颈。

8.5 总结

本章由浅入深地介绍了三种构造对话机器人的方法，其中有两种属于索引式模型，一种属于最新的生成式模型。第一种是基于深度学习流行之前的技术，使用 AIML 这种标识语言构造大量的应答库，通过现有的对文字结构的理解来构造的简单对话系统。这种系统的构造费时费力，灵活性差，扩展性差，智能度低，很难构造多轮对话系统，但是因为应答都是真人生成的，不会有语法错误，语言标准，所以适用于简单集中的业务环境。第二种是使用深度学习方法来寻找对应于当前对话背景的最佳应答，相对于第一种方法降低了很多人工构造应答库的工作，灵活性高，扩展性强，有一定智能度，可以用来构造多轮对话系统。第三种是目前最新研究的领域，使用深度学习技术实时生成应答，灵活度和智能度都极高，属于自动扩展，但是需要极大量的数据积累和比较复杂的模型才能得到较好的结果。通常第三种系统需要与第二种系统相结合，在第二种系统已有的应答库中无法找到足够满意的选项时，可以启用第三种系统来实时生成应答。

9 时间序列

9.1 时间序列简介

时间序列是在商业数据或者工程数据中经常出现的一种数据形式，是以时间为次序排列，用来描述和计量一系列过程或者行为的数据的统称。比如每天商店的收入流水或者某个工厂每小时的产品产出都是时间序列数据。一般研究的时间序列数据有两种类型。最常见的是跟踪单一的计量数据随时间变化的情况，即每个时间点上收集的数据是一个一维变量，这种是最常见的，通常的时间序列默认就是这种数据，也是本章研究的对象。另外一种时间序列数据是多个对象或者多个维度的计量数据随时间变化的情况，即每个时间点上收集的数据是一个多维变量，这种一般也被称为纵向数据（Longitudinal Data），但是不属于本章研究的对象。

本章首先介绍几个与时间序列相关的基本概念，比如平稳性（Stationarity），随机行走（Random Walk）等；其次介绍我们数据的示例；然后还会介绍深度学习中的循环神经网络（RNN）模型及其变形的长短期记忆人工神经网络（LSTM），这类模型是在实践中将深度学习技术应用于时间序列数据最常见的模型。最后将 LSTM 应用于示例数据并进行分析和预测，以及展示实际效果。本章以实践为主，强调具体概念和模型的应用，希望读者在读完本章以后能将其快速应用到自己的工作中。

为了便于下面的程序执行，这里先将需要的软件库提前载入当前系统。这些常用的软件库包括 Pandas、Numpy、Matplotlib 以及 StatsModels。

```
%matplotlib inline
import pandas as pd
import numpy as np
import statsmodels.api as sm
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
plt.rcParams['figure.figsize']=(20, 10)
```

我们可以通过如下命令检查 StatsModels 的版本号：

```
sm.version.full_version
```

屏幕上显示当前所用的版本为 0.8.0rc1。如果读者现在安装 StatsModels 包，则应该已经是正式的 0.8.0 版本。

9.2 基本概念

有效的时间序列分析依赖于几个核心概念。熟悉掌握这些核心概念能帮助分析师在实际面对数据的时候能有效地切入。

其中最核心的概念是平稳性（Stationarity）。我们在分析时间序列数据时，需要考虑这个时间序列反映的随机过程是否稳定。如果一个时间序列不稳定，则说明其来自于的总体在发生变化，那么在忽略这种情况下进行的分析并不有效，特别是不能有效地应用于对未来事件的预测。时间序列数据 $y_t, t = 1, \cdots, T$ 的稳定性定义有很多种角度，其中使用最广泛的就是数学上讲的弱平稳性，其定义如下：

y_t的期望值$E(y_t)$不是时间t的函数：

$$E(y_t) = \mu$$

y_s和y_t之间的协方差只是时间单位差绝对值$|s-t|$的函数：

$$\text{Cov}(y_s, y_t) = \text{Cov}(y_{s+z}, y_{t+z})$$

具体来讲就是，在弱平稳性的假设条件下，期望值不依赖于时间而变化，比如 $E(y_4) = E(y_8)$；而协方差只是两个序列时间间隔的区间的函数，比如 $\text{Cov}(y_4, y_6) = \text{Cov}(y_7, y_9)$，因为 y_4, y_6 和 y_7, y_9 一样只间隔两个时间点。第二条假设隐含的意思就是弱平稳性的时间序列方差恒定（Homoscedasticity），即 $\text{Cov}(y_t, y_t) = \text{Cov}(y_s, y_s) = \sigma^2$。

比弱平稳性更强的数学假设条件为强平稳性，也称为严格平稳性，这一假设条件下要求随机变量 y_t 的整个概率分布不随时间的改变而变化。但是在一般的应用场景下，满足弱平稳性条件已经能够适用于大多数模型。

第二个概念为白噪声（White Noise）。白噪声是研究随机过程中经常出现的概念，是联系横截面数据（Cross Sectional Data）和纵向数据的纽带。严格来讲，白噪声是具有独立同分布（i.i.d）的数据序列，即没有特定随时间变化特征的满足平稳性条件的数据，当然，满足平稳性条件的数据类型有很多，白噪声只是其中一种，另外一种满足平稳性条件的时间序列数据类型就是我们在本章也要提到的自回归过程。白噪声数据在时间序列研究中之所以重要是因为所有时间序列的技术都是要将一组数据通过一系列过程尽量变为一个白噪声数据，这一系列过程就被称之为滤子。

白噪声数据的特点是对其的点预测及其方差不依赖于我们想要预测到多远，而只与样本数据的均值和方差有关。举例来讲，如果我们有个白噪声过程 $y_t, t = 1, \ldots, T$，而我们要预测 $T + s$ 期未来数据的大小，则其最优期望值为样本均值 $\bar{y}$，而预测的 α 置信区间为

$$\bar{y} \pm t_{T-1,1-\alpha/2}\sqrt{(1 + 1/T)}s_y$$

其中 s_y 为样本方差根，而 $t_{T-1,1-\alpha/2}$ 则是自由度为 $T - 1$ 的 T-分布统计量 α 百分位下的对应值，通常 95% 百分位下大约为 2。

第三个概念是随机行走（Random Walk）。白噪声时间序列的累加和就构成一个随机行走时间序列。举例来讲，如果 $z_t, t = 1, \cdots, T$ 是一组白噪声序列，则 $y_t = \sum_1^t z_t, t \geqslant 1$ 构成一组随机行走序列。图9.1演示了一个均值为 0.1，标准差为 2 的 100 个时间点的白噪声，及其对应的随机行走时间序列。图9.1由以下程序生成。

```
np.random.seed(1291)
z = np.random.normal(0.1, 2, 100)
y = np.cumsum(z)

fig, ax1 = plt.subplots()
plt.plot(z, label="White Noise")
plt.plot(y, label="Random Walk")
plt.legend()
plt.show()

mean1 = np.round(np.mean(y[:20]), 4); mean2=np.round(np.mean(y[-20:]), 4);
std1 = np.round(np.std(y[:20]), 4); std2=np.round(np.std(y[-20:]), 4)

```

```
print("前20个数据点的均值为%.4f, 标准差为%.4f" %(mean1, std1))
print("\\")
print("后20个数据点的均值为%.4f, 标准差为%.4f" %(mean2, std2))
```

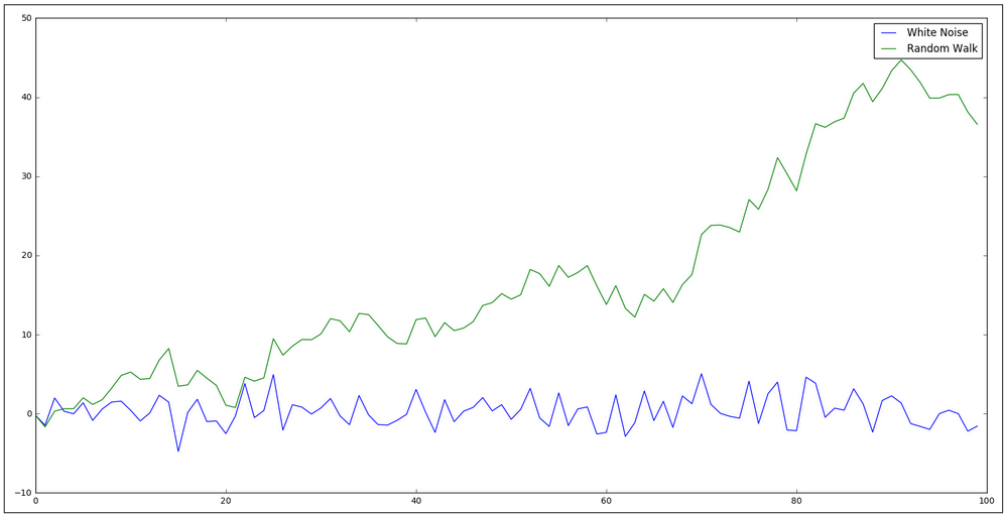


图 9.1　白噪声和随机行走例图

从图9.1中可以看到随机行走时间序列的几个特点。首先这种时间序列数据是非平稳的，其均值和方差都随着时间而变化。比如这个时间序列前 20 个时间点的均值为 3.13，而最后 20 个时间点的均值为 38.97；其对应的标准差则分别为 2.45 和 3.76，对这个随机行走时间序列取一阶差分作为滤子，过滤后的时间序列则为上例中的白噪声序列。

随机行走模型是一类非常重要的时间序列模型。因为其为对应的白噪声时间序列的累加和，所以每个时间点上该变量的期望和方差分别为：

$$E(y_t) = y_0 + t\mu$$
$$\mathrm{Var}(y_t) = t\sigma^2$$

其中 μ, σ^2 是对应的白噪声序列的期望均值和方差，而 y_0 则为这个白噪声随机变量在初始时间的某个具体实现。只要这个均值大于 0，则随机行走时间序列表现为总体上一个不断增长的曲线，由图9.1看出；而如果这个均值小于 0，则随机行走时间序列表现为一个总体上不断下降的曲线。另外，随机行走时间序列的方差也是时间的线性函数。可见随机行走模型是一个随时间变动的线性模型。相应地，如果要对一个随机行走时间序列进行预测，则其公式为：

$$y_{T+s} = y_T + s\hat{\mu} \pm 2\hat{\sigma}\sqrt{s}$$

其中，y_T 是已知随机行走时间序列的末尾值，s 是要预测的未来时间间隔，$\hat{\mu}, \hat{\sigma}$ 则分别是对应的白噪声过程的期望均值和标准差的估计值，通常为样本的均值和标准差。

可以看到，对于白噪声和随机行走两种不同的时间序列的预测有不同的模型，那么怎么识别一个已知的时间序列是平稳的还是一个随机行走时间序列呢？

首先，如果要识别一个时间序列是否是平稳的，通过检验单位根的方法，常用的有以下几种，在 Python 的 StatsModels 里面都有现成的函数可用。

Augmented Dickey-Fuller Test (ADF)

(1) ADF 是最常见的单位根检验方法。其默认假设待验证的时间序列是不平稳的，如果得到的统计量的 p 值较大，则说明这个时间序列是不平稳的，如果 p 较小，则说明这个时间序列是平稳的。假如我们用 5% 作为 p 值的界限，那么如果 ADF 统计量的 p 值大于 0.05 则表明时间序列是不平稳的，需要做差分运算，一直到检验结果表明是平稳的为止。

(2) 在 Python 中我们可以用 StatsModels 软件库里的 tsa.stattools.adfuller(x) 函数来检验时间序列 X 的平稳性。

Kwiatkowski-Phillips-Schmidt-Shin Test (KPSS)

(1) KPSS 检验是一种较新的检验方式，与 ADF 检验相反，其默认假设待验证的时间序列是平稳的，如果得到的统计量 p 值较大，则说明这个时间序列是平稳的；反之则是不平稳的。

(2) 在 Python 中可以用 StatsModels 软件库里的 tsa.stattools.kpss(x) 函数来检验时间序列的平稳性。注意，kpss.test 这个函数只在 StatsModels 0.8 以上版本才有。

StatsModels 的版本可以通过以下命令查阅：

```
1 import statsmodels
2 print(statsmodels.version.full_version)
```

如果现有系统不是这个版本的 statsmodels，可以通过 pip 升级：

```
pip install statsmodels=0.8.0rc1
```

9.3 时间序列模型预测准确度的衡量

时间序列模型通常用来对未来的值进行预测，那么衡量预测的值的准确性就很重要了。下面我们先简要介绍检验预测模型的一些常用统计量，然后介绍使用样本外数据验证步骤。

衡量预测准确度的常用统计量

(1) 平均误差（Mean Error，ME）：

$$\mathrm{ME} = \frac{1}{T_2}\sum_{t=T_1+1}^{T_1+T_2} e_t$$

平均误差能较好地衡量现有模型是否有很好描述的线性趋势。

(2) 平均百分比误差（Mean Percentage Error，MPE）：

$$\mathrm{MPE} = \frac{1}{T_2}\sum_{t=T_1+1}^{T_1+T_2} \frac{e_t}{y_t}$$

平均百分比误差也用于衡量是否有短期趋势没有被模型很好地描述，不过它是以相对误差的形式来表达的。

(3) 均方差（Mean Square Error，MSE）：

$$\mathrm{MSE} = \frac{1}{T_2}\sum_{t=T_1+1}^{T_1+T_2} e_t^2$$

均方差相对于平均误差来说，能侦测出线性趋势之外更多的没有被模型描述的数据模式，比如周期性等，因此其更为常用。

(4) 平均绝对误差（Mean Absolute Error，MAE）：

$$\mathrm{MAE} = \frac{1}{T_2}\sum_{t=T_1+1}^{T_1+T_2} \|e_t\|$$

平均绝对误差在衡量模型的准确度方面和均方差有类似的效果，只是对于异常值相对来说稳健性更高。

(5) 平均绝对百分比误差（Mean Absolute Percentage Error，MAPE）：

$$\text{MAPE} = \frac{1}{T_2}\sum_{t=T_1+1}^{T_1+T_2} \| \frac{e_t}{y_t} \|$$

MAPE 结合了 MAE 和 MPE 的优点，能较好地侦测线性趋势之外的更多的数据模式，并以相对误差的形式表达。

使用样本外数据验证步骤

(1) 将长度为 $T = T_1 + T_2$ 的样本时间序列分为两个子序列，其中前面一个（$t = 1, \cdots, T_1$）子序列用于模型训练，后面一个子序列（$t = T_1 + 1, \cdots, T$）用于模型验证。

(2) 用第一个子序列训练一个待验证模型。

(3) 使用上一步训练的模型，使用时间范围为 $t = 1, \cdots, T_1$ 的因变量来预测未来 $T_1 + 1, \cdots, T$ 时间段的因变量值：$\hat{y}_t$，即对用于模型验证部分的子序列因变量使用待验证模型进行拟合。

(4) 使用上一步拟合的因变量值和对应的实际因变量值，计算单步预测误差：$e_t = y_t - \hat{y}_t$，然后采用一种或者多种9.2节介绍的衡量模型准确度的统计量来计算综合预测能力。

可以对每一个待验证模型都执行第（2）到第（4）步，选取综合预测能力最好，即统计量值最小的那个待选模型。

9.4 时间序列数据示例

我们的时间序列数据来自于DataMarket的时间序列数据库：`https://datamarket.com/data/list/?q=provider:tsdl` 。这个库由澳大利亚莫纳什大学的统计学教授 Rob Hyndman 创建，收集了数十个公开的时间序列数据集。本章我们采用其中两个数据作为实例。Rob Hyndman 教授也是 R 统计语言里面 forecast 软件包的开发者。

第一个数据是在汉口测量的长江每月流量数据，其文件名为 monthly-flows-chang-jiang-at-han-kou.csv，读者可以到`www.broadview.com.cn/31872`下载。该数据记录了从 1865 年 1 月到 1978 年 12 月在汉口记录的长江每月的流量，总计 1368 个数据点，计量单位未知，不过这不妨碍我们的分析过程和结果。我们将该数据下载后在本地磁盘存为：E:\data\TimeSeries\monthly-flows-chang-jiang-at-hankou.csv。

```
parser = lambda date: pd.datetime.strptime(date, '%Y-%m')
df1 = pd.read_csv("e:/data/timeseries/monthly-flows-chang-jiang-at-hankou.
csv", engine="python", skipfooter=3, names=["YearMonth", "WaterFlow"],
parse_dates=[0], infer_datetime_format=True, date_parser=parser, header=0)
print(df1.head())
df1.YearMonth = pd.to_datetime(df1.YearMonth)
df1.set_index("YearMonth", inplace=True)
df1.plot()
plt.show()
```

从图9.2可以看出，该数据具备很强的不同长度的周期性。

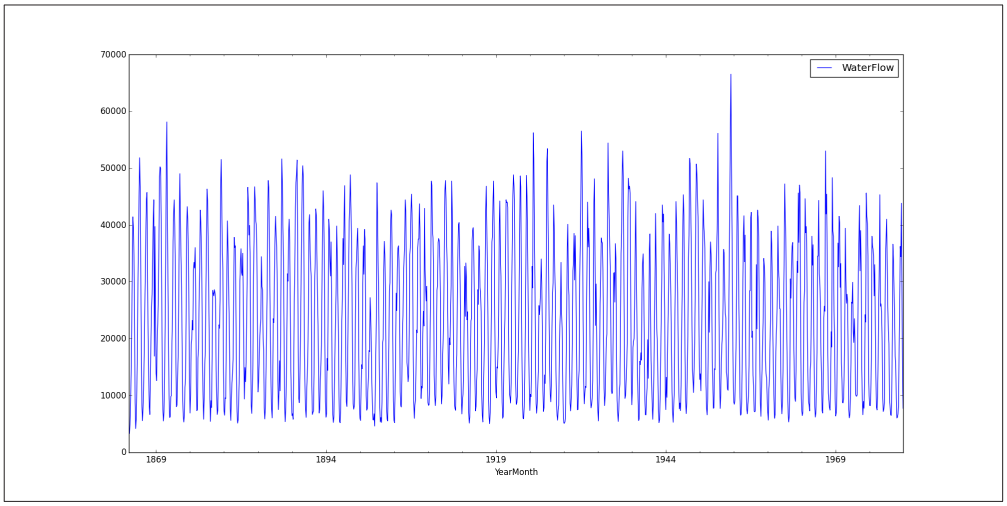


图 9.2　长江汉口水流量历史数据

第二个数据是从 1949 年 1 月到 1960 年 12 月的月度国际航空旅客数量，其文件名为 international-airline-passengers.csv，读者可以到 www.broadview.com.cn/31872 下载。该数据有 144 个数据点，数据单位为千人，与第一个数据不同的是，该数据包含极强的趋势要素和周期要素，因此在具体的分析上能体现不同的要求。该数据下载后在本地磁盘存为：E:\data\TimeSeries\international-airline-passengers.csv。下面读入该数据并展示，如图9.3所示。

```
parser = lambda date: pd.datetime.strptime(date, '%b-%y')

df2 = pd.read_csv("e:/data/timeseries/international-airline-passengers.csv",
 engine="python", skipfooter=3, names=["YearMonth", "Passenger"],header=0)
df2.YearMonth = df2.YearMonth.str[:4]+'19'+df2.YearMonth.str[-2:]
```

```
df2.YearMonth = pd.to_datetime(df2.YearMonth, infer_datetime_format=True)
df2.set_index("YearMonth", inplace=True)
print(df2.head())
df2.plot()
plt.show()
```

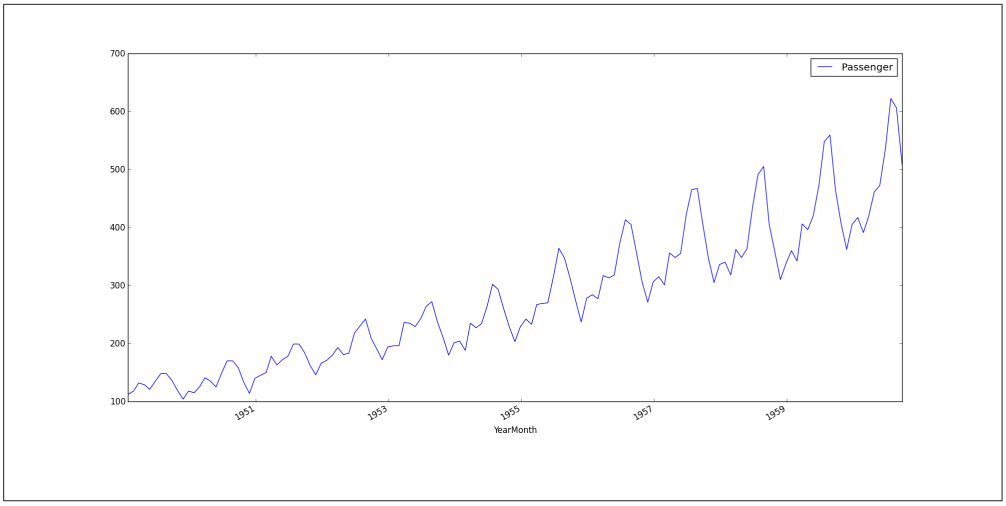


图 9.3 国际航空旅客数量

9.5 简要回顾 ARIMA 时间序列模型

在我们讲解循环神经网络算法之前，我们先简要回顾一下传统的 ARIMA 时间序列模型，对上述数据进行常规建模和预测。在9.7节会将 ARIMA 的预测结果与神经网络模型的预测结果进行比较。

ARIMA 模型即自回归积分移动平均（Auto Regressive Integrated Moving Average）模型，ARIMA 模型通常写作 ARIMA（*p,d,q*），其中：

(1) p 指自回归项的个数，是使用取差分平稳化以后的新时间序列的过去值作为解释变量部分的个数。

(2) d 指将序列平稳化所需的差分次数，反过来，从平稳化的序列变化为原始数据的算法即称为预测方程。假如原始数据为 Y_t，而差分后的平稳数据为 y_t，如果 $d=0$，则 $Y_t=y_t$，如果 $d=1$，则 $Y_t=y_t+Y_{t-1}$，而如果 $d=2$，则 $Y_t=(y_t+Y_{t-1})+(Y_{t-1}-Y_{t-2})$。

(3) q 对应移动平均部分，指预测方程里预测误差的滞后项个数。

这是一类非常灵活的时间序列预测模型，通常使用在可以通过差分变换为平稳序列的时间序列数据上。在这里要强调一下，在对时间序列数据进行平稳化的过程中，我们通常也一起使用对数或者 Box-Cox 变换等手段。（弱）平稳序列的含义是指这个数据没有特定的趋势，并且其围绕其平均值按照比较一致的波幅进行波动。这个波幅一致的波动意味着其自相关系数不随时间而变化，或者说其功率频谱不变。这种时间序列数据可以被看成一个信号和一个噪声项的组合，其信号项部分可以是一个或者多个往复的三角函数曲线以及其他周期性信号的组合。从这个角度讲，ARIMA 模型可以被看做一个试图将信号与噪声分离的滤子，并使用外推法预测未来值。

ARIMA 模型的一般形式写作：

$$\hat{y}_t = \mu + \alpha_1 y_{t-1} + \cdots + \alpha_p y_{t-p} + \beta_1 e_{t-1} - \cdots - \beta_q e_{t-q}$$

使用 ARIMA 模型建模的步骤如下：

(1) 可视化待建模的序列数据。

(2) 使用 ADF 或者 KPSS 测试确定将数据平稳化所需的差分次数。

(3) 使用 ACF/PACF 确定移动平均对应的预测误差项和自回归项个数，一般从一项开始。

(4) 对于拟合好的 ARIMA 模型，将预测误差项和自回归项分别减少一个再拟合。

(5) 根据 AIC 或者 BIC 判断模型相对简单的 AR 或者 MA 模型是否有改进。

(6) 对自回归和移动平均项个数递增一个，逐次检验。

对于自相关性，一般可以通过增加自回归项或者移动平均部分里面的预测误差项个数来消除。一般的原则是如果未消除的自相关是正自相关关系，即 ACF 图里面第一项是正值，则使用增加自回归项的方法较好；而如果未消除的自相关是负自相关关系，则增加预测误差项的方法更为合适。这是因为一般而言，差分方法对于消除正相关关系非常有效，但是同时也会额外引入反向的相关关系，这时候会出现过度差分的情况，需要额外引入一个预测误差项来消除负相关关系，这也是为什么在上面的建模步骤里面先引入预测误差项建模，而不是先引入自回归项开始建模，也就是先拟合一个 ARIMA(0，1，1) 模型再看看 ARIMA(1，1，0) 模型，通常 ARIMA(0，1，1) 模型会比 ARIMA(1，1，0) 模型拟合效果好一些。

下面是杜克大学 Fuqua 商学院的 Robert F. Nau 教授总结的 13 项 ARIMA 模型建模需要遵守的一般原则。

识别差分项的原则

(1) 如果建模的序列正的自相关系数一直衍生到很长的滞后项（比如 10 或者更多滞后项），则获得平稳序列所需的差分次数较多。

(2) 如果滞后一项的自相关系数为 0 或者为负，或者所有的自相关系数都很小，则该序列不需要更多的差分来获取平稳性。通常而言，如果滞后一项的自相关性为-0.5 或者更小，则很可能该序列被过度差分了，这是需要注意的。

(3) 最优的差分项个数通常对应于差分后拥有最小标准差的时间序列。

(4) 如果原序列不需要进行差分，则假定原序列是平稳的。一阶差分则意味着原序列有一个为常数的平均趋势。二阶差分则意味着原序列有一个依时间变化的趋势。

(5) 对不需要进行差分的时间序列建模时通常包含一个常数项。如果对一个需要一阶差分的时间序列进行建模，则只有在该时间序列包含非 0 的平均趋势的时候才需要包含常数项。而对一个需要进行二阶差分的时间序列进行建模时则通常不用包含常数项。

识别自回归或者预测误差项的原则

(1) 如果差分后的序列的 PACF 显示为 Sharp Cutoff 或者滞后一项的自相关为正，则说明该序列差分不足，这时候可以对模型增加一个或者多个自相关项，增加个数通常为 PACF Cutoff 的地方。

(2) 如果差分后的序列的 ACF 显示为急剧截断或者滞后一项的自相关为负自相关，则说明该序列差分过度，这时候可以对模型增加一个或者多个预测误差项，增加个数通常为 ACF 截断（Cutoff）的地方。

(3) 自回归项和预测误差项有可能会互相抵消，因此如果一个两种要素都包含的 ARIMA 模型对数据拟合得很好，则通常可以试一试一个少一个自回归项或者少一个预测误差项的模型。一般来说，同时包含多个自回归和多个预测误差项的 ARIMA 模型都会过度拟合。

(4) 如果自回归项的系数和接近 1，即自回归部分有单位根现象，那么这时候应该将自回归项减少一个，同时增加一次差分操作。

(5) 如果预测误差项的系数和接近 1，即移动平均部分有单位根现象，那么这时候应该将预测误差项减少一个，同时减少一次差分操作。

(6) 自回归或者移动平均部分有单位根通常也表现为长期预测不稳定。

识别模型的季节性

(1) 如果一个时间序列有很强的季节性，则必须使用一次季节周期作为差分，否则模型会认为季节性会随着时间逐渐消除。但是使用季节周期做差分不能超过一次，如果使用了季节周期做差分，则非季节周期的差分最多也只能再进行一次。

(2) 如果一个适当差分之后的序列的自相关系数在第 s 个滞后上仍然表现为正，而 s 为季节性周期包含的时间段数，则在模型里添加一个季节性自回归项。如果这个自相关系数为负，则添加一个季节性预测误差项。通常情况下，如果已经使用了季节周期做差分，则第二种情况更常见（见前面对自相关性处理的解释），而第一种情况通常是还没使用季节性周期做差分。如果季节性周期很规律，则使用差分是比引入一个季节性自回归项更好的方法。分析师应该尽量避免在模型里同时引入季节性自回归和季节性预测误差项，否则模型会过度拟合，甚至在拟合过程本身会出现不收敛的情况。

下面使用国际航空旅客数量为例子来展示 ARIMA 模型的建模过程。使用这个数据是因为从图像上看这个数据具有很强的趋势性和周期性，因此能够充分展示建模的不同步骤。

首先确定所需的差分阶数，这可以通过对不同阶数依次差分的序列进行 ADF, KPSS 检验和检查 ACF, PACF 图来实现。因为从图像上看该序列具有很强的趋势性，并且数值波动范围不停增大，即具有异方差性，所以我们先对该序列取对数将异方差变为同方差，再从变换以后的序列的一阶差分开始检验。

需要注意的是，KPSS 测试在 p 值过小或者过大的情况下，会打印“警告”（warnings）信息。这是一个非常差的设计。出于排版美观的要求，在下面的检验中，使用了 warnings 软件库来对 KPSS 函数的 warnings 打印信息进行控制，直接忽略不打印。

```
order=1
diff1 = df2.Passenger.diff(order)[order:]
logdiff1 = np.log(df2.Passenger).diff(order)[order:]
adftest = sm.tsa.stattools.adfuller(diff1)
adftestlog = sm.tsa.stattools.adfuller(logdiff1)
print("ADF test result on Difference shows test statistic is %f \
and p-value is %f" %(adftest[:2]))
print("ADF test result on Log Difference shows test statistic is %f \
and p-value is %f" %(adftestlog[:2]))

import warnings
with warnings.catch_warnings():
```

```
    warnings.filterwarnings("ignore")
    kpsstest = sm.tsa.stattools.kpss(diff1)
    kpsstestlog=sm.tsa.stattools.kpss(logdiff1)

print(\"KPSS test result on Difference shows test statistic is %f \
and p-value is %f\" %(kpsstest[:2]))
print("KPSS test result on Log difference shows test statistic is %f \
and p-value is %f" %(kpsstestlog[:2]))
```

结果如下所示。

```
ADF test result on Difference shows test statistic is -3.045022 and p-value
is 0.030898
ADF test result on Log Difference shows test statistic is -2.706950 and p-
value is 0.072843
KPSS test result on Difference shows test statistic is 0.078160 and p-value
is 0.100000
KPSS test result on Log difference shows test statistic is 0.059560 and p-
value is 0.100000
```

从检验结果来看，除了 ADF 检验对原数据直接取一阶差分的情况刚好通过平稳性检验，KPSS 检验两个差分数据都没通过平稳性检验，而对数差分数据没有通过 ADF 检验。

下面看一看图9.4展示的 ACF 和 PACF 情况。

```
fig, ax = plt.subplots()
ax1=fig.add_subplot(221)
sm.graphics.tsa.plot_acf(diff1, ax=ax1)

ax2=fig.add_subplot(222)
sm.graphics.tsa.plot_pacf(diff1, ax=ax2)

ax3 = fig.add_subplot(223)
sm.graphics.tsa.plot_acf(logdiff1, ax=ax3)

ax4=fig.add_subplot(224)
sm.graphics.tsa.plot_pacf(logdiff1, ax=ax4)

plt.show()
```

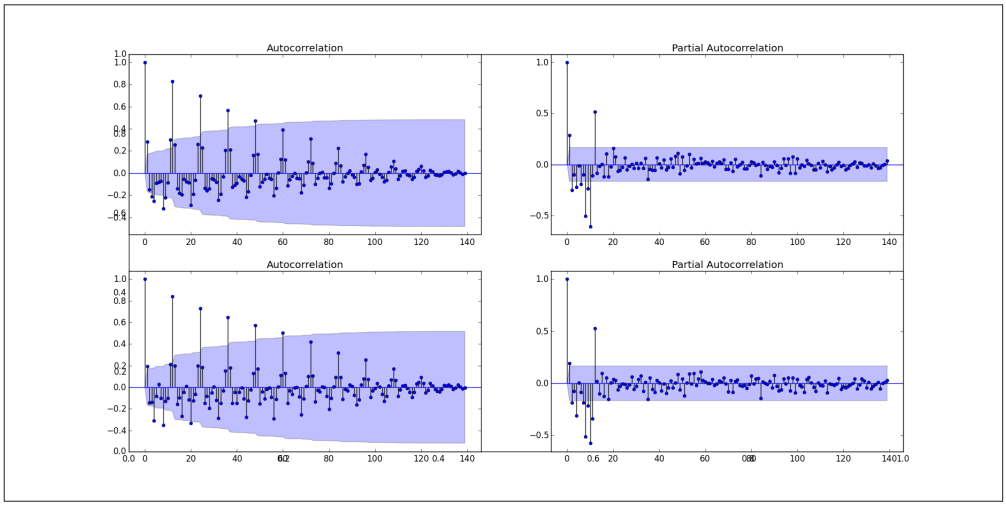


图 9.4　一阶差分的平稳性检验

ACF 和 PACF 的结果有以下几个值得注意的地方。

(1) 这个时间序列数据有很强的周期性，周期大约为 12 个月，这非常符合自然的经济解释，需要引入一次季节性差分操作。

(2) 根据 PACF 的结果，无论是原数据的一阶差分，还是对数数据的一阶差分，都需要再次进行一次差分操作，并对二阶差分的模型引入一个自回归项。

(3) 根据 ACF 的结果，在没有过度差分的情况，可以分别测试带一个预测误差项的 ARIMA 模型和不带预测误差项的 ARIMA 模型。

9.6　循环神经网络与时间序列模型

传统时间序列模型和循环神经网络模型有着很密切的联系。不论是自回归模型还是移动平均自回归模型，均可以看作 RNN 模型的一种特例。下面详细解释。

自回归模型可以用如图9.5所示的 RNN 模型图例来表示。

图9.5中使用的是 RNN 的语言，但是如果翻译一下就会发现其是标准的 AR 模型的延伸。比如，h 对应自回归模型里面的待预测变量，即状态变量，而 X_t 是当期的输入层信息，在自回归模型里就是当期的预测误差 ϵ_t。这里为了不至于让读者产生困惑，在这个模型里，在不考虑误差的情况下，状态变量的动态可以表示为如下的数学公式：

$$h_t = \phi(\boldsymbol{W}_{xh}x_t + \boldsymbol{W}_{hh}h_{t-1} + b)$$

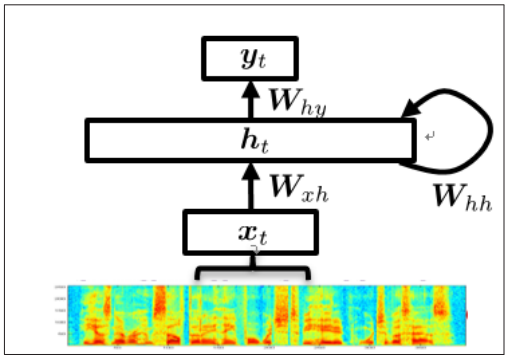


图 9.5 AR(p) 模型在 RNN 结构中的示意图（图片来源：Hybrid Deep Neural Network--Hidden Markov Model (DNN-HMM) Based Speech Emotion Recognition）

其中，$\boldsymbol{W}_{xh}$ 表示输入层的权重矩阵，而 $\boldsymbol{W}_{hh}$ 表示隐藏层回馈权重，b 则是偏移项，在标准回归模型中通常被称为截距项，$\phi()$ 是施加在隐藏层单元上的非线性函数，比如常用的 Sigmod 函数等。如果将上述的权重矩阵、状态变量和函数约束并按照统计语言改写一下，即规定 $\boldsymbol{W}_{hh} = \beta, \boldsymbol{W}_{xh} = \alpha, h_t = y_t$，同时将 $\phi()$ 规定为 Identity 函数，则上述公式变为：

$$\hat{y}_t = b + \alpha y_{t-1} + \beta x_t$$

这与标准的带外生变量的一阶自回归模型没什么区别了。使用 RNN 做最后预测的时候，则使用输出权重矩阵 $\boldsymbol{W}_{hy}$ 按照如下公式进行：

$$y_t = \zeta(\boldsymbol{W}_{hy} h_t)$$

$\zeta()$ 是输出层的非线性函数，常用的有 Softmax，tanh 等。但是在自回归模型里，隐藏状态变量即是要预测的变量期望，即 $y_t = E\hat{y}_t$，因此相应的 $\zeta()$ 变为 Identity 函数，而 $\boldsymbol{W}_{hy}$ 也溃缩为单位 1。

上面的结构可以很自然地延伸到自回归移动平均 (ARMA) 模型。在对照自回归移动平均模型的 RNN 模型里，隐藏层状态变量的动态可以表示为：

$$h_t = \phi(\sum_{j=\delta_1}^{\delta_2} \boldsymbol{W}_{xh,j} x_{t-j} + \boldsymbol{W}_{hh} h_{t-1} + b)$$

和标准的 ARMA 模型不同，在上述 RNN 模型里，模型可以往前看 δ_1 期的样本，而在标准的模型中只能往后看，即 $\delta_1 = 0, \delta_2 > 0$。同样地，按照前面对 AR 模型的处理，用相应的标准统计语言来替代神经网络模型的符号，则上述公式变为：

$$y_t = b + \alpha y_{t-1} + \sum_{j=1}^{q} \theta_j e_{t-j} + \beta x_t$$

这是一个标准的 ARMA(1,q) 模型。这个结构可以通过图9.6来表示。

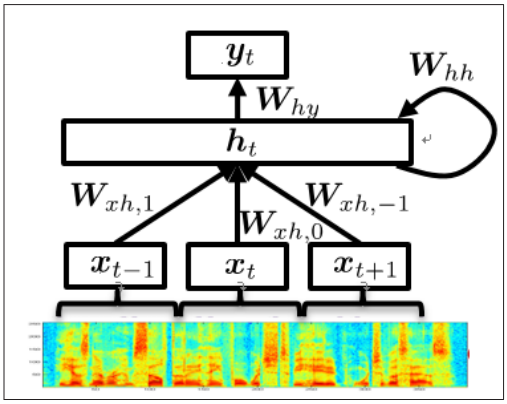


图 9.6　ARMA(p,q) 模型在 RNN 结构的示意图（图片来源：Hybrid Deep Neural Network--Hidden Markov Model (DNN-HMM) Based Speech Emotion Recognition）

ARMA 中的预测误差在 RNN 模型中被作为输入层信息，虽然在模型估计阶段预测误差并不能被获知。通常 ARMA 模型是通过最大似然法（MLE）来拟合，不过，ARMA(p, q) 模型也可以通过两阶段线性回归来拟合，这个拟合过程可以帮助读者理解为何在 RNN 模型中预测误差被作为输入层信息。在两阶段线性回归中，首先是拟合一个 AR(p) 模型，根据这个模型的预测值得到预测误差的序列，然后对于每一期数据引入 q 期滞后的预测误差作为回归变量。可见，将预测误差作为输入层信息是完全合理、自然的选择。

综上所述，RNN 模型和传统的时间序列模型有非常深刻的联系，RNN 模型可以被看作对传统模型在模型向量维度、时间维度以及函数形式上的延伸，而隐藏层的状态变量在传统时间序列模型中就是指真实的数据生成过程（DGP）所对应的待预测变量，而外生变量和预测误差都是作为输入层信息进入 RNN 模型的。

9.7　应用案例

本节应用前面学到的理论对真实的时间序列数据进行建模并预测。这里会用到前面提到的长江汉口地区月度流量数据和全球航空公司月度乘客数量两个时间序列数据，具体介绍如何应用 ARIMA 模型和 LSTM 模型对时间序列数据进行建模和预测，并对

两种模型的实际预测能力进行比较。在使用 ARIMA 模型进行建模的时候，按照下面的标准步骤操作。

(1) 首先对示例数据进行标准分析，包括识别其平稳性以及是否是随机行走或者具备单位根问题。

(2) 使用周期图法（Periodogram）来识别季节性。

(3) 对于去除季节性的数据，通过 ACF 和 PACF 函数提供的信息得到自回归和移动平均部分所需的滞后项个数。这些数据用于对 ARIMA（p, d, q）模型的参数进行标定。

(4) 然后对时间序列数据进行拟合，并得到相应的检验量。

(5) 对残差检验 Q 统计量和 JB 统计量以及计算 ACF 和 PACF 函数，确定没有额外的信息游离于模型之外。

(6) 最后对样本外的测试数据进行预测并检验模型的准确度。

在使用 LSTM 模型对时间序列建模的时候，我们依然需要进行前面的第（1）和第（2）步和第（4）步，但是中间的步骤略有不同。

(1) 首先对示例数据进行标准分析，包括识别其平稳性以及是否是随机行走或者具备单位根问题。

(2) 使用周期图法（Periodogram）来识别季节性。

(3) 对于去除季节性的数据，通过 ACF 和 PACF 函数提供的信息得到建模所需包含的滞后项个数，包括在 LSTM 模型中需要将多久以前的信息带入当期。

(4) 对数据进行处理，使其符合 Keras 软件包的 LSTM 模型 API 的要求。

(5) 拟合多种不同结构的 LSTM 模型并比较其在样本内数据上的表现。

(6) 对残差计算 ACF 和 PACF 函数，确定没有额外的信息游离于模型之外。

(7) 最后对样本外的数据进行预测，并检验其表现。

我们先载入数据分析和建模所需的软件库。

```
import Keras.models as kModels
import Keras.layers as kLayers
from scipy.signal import periodogram
import warnings

from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
```

在我们的案例中，信息显示我们使用 GPU 作为计算核心，同时启动了 cuDNN 库。这里不再展示了。

9.7.1 长江汉口月度流量时间序列模型

在第一例子里，我们对长江流量的月度数据进行建模，先建立一个 ARIMA 模型，再建立一个基于 LSTM 的深度学习模型，最后我们比较两个模型的预测性能。在建模之前，我们先将数据分为训练集合和样本外测试集。我们将最后 24 个月的数据留作测试集，其余的作为训练集。

```
cutoff=24
train = df1.WaterFlow[:-cutoff]
test = df1.WaterFlow[-cutoff:]
```

作为数据分析的第一步，首先我们检验这组数据是否平稳，以及分析其是否需要进行相应的操作以获得平稳性。根据前面提到的平稳性检验方法，我们可以通过观测移动平均和移动均方差随时间的变化图，以及正式的 Dicky-Fuller 和 KPSS 检验来实现。下面我们构造一个函数，将这些功能都集成在里面。

```
def test_stationarity(timeseries, window=12):
    import statsmodels.api as sm
    import pandas as pd
    df = pd.DataFrame(timeseries)
    df['Rolling.Mean'] = timeseries.rolling(window=window).mean()
    df['Rolling.Std']=timeseries.rolling(window=window).std()
    adftest = sm.tsa.stattools.adfuller(timeseries)
    adfoutput = pd.Series(adftest[0:4], index=['统计量','p-值  ','滞后量','
    观测值数量'])
    for key,value in adftest[4].items():
        adfoutput['临界值 (%s)'% key] = value
    return(adfoutput, df)
```

下面我们对整个训练集和训练集的局部执行上述函数检验平稳性，如图9.7所示。

```
fig = plt.figure()
ax0 = fig.add_subplot(221)
adftest, dftest0=test_stationarity(train)
dftest0.plot(ax=ax0)
```

```
print('原始数据平稳性检验')
print(adftest)

ax1 = fig.add_subplot(222)
adftest, dftest1=test_stationarity(train['1960':'1975'])
dftest1.plot(ax=ax1)
print('局部数据平稳性检验')
print(adftest)
```

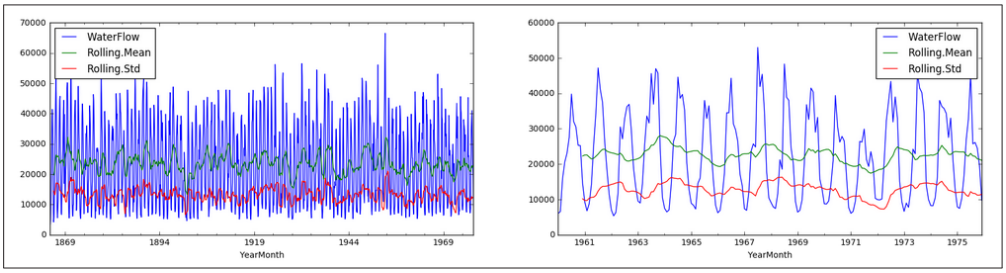


图 9.7 平稳性检验

我们发现，虽然 Dicky-Fully 检验的统计量显著表明，不论是整个训练集还是所选取的局部，对于统计测试都显得是平稳的，但是仔细观察移动平均值和移动均方差，发现其波动幅度还是很大的，并且有非常强的季节性。下面我们需要确定季节性的周期以便构造 ARIMA 模型。对于季节性时间序列，我们会构造 SARIMA 模型，对季节性周期做相应的处理。按照前面提到的方法，侦测季节性周期可以使用 ACF 函数，对原始数据的 ACF 序列计算周期图法（Periodogram），通常在周期的时间点上有高度的能量集中，从而能够识别周期长度。下面这个函数展示了这种方法。

```
def CalculateCycle(ts, lags=36):
    import statsmodels.api as sm
    from statsmodels.tsa.stattools import acf
    from scipy import signal
    import peakutils as peak
    acf_x, acf_ci = acf(ts, alpha=0.05, nlags=lags)
    fs=1
    f, Pxx_den = signal.periodogram(acf_x, fs)

    index = peak.indexes(Pxx_den)
    cycle=(1/f[index[0]]).astype(int)

```

```
    fig = plt.figure()
    ax0 = fig.add_subplot(111)
    plt.vlines(f, 0, Pxx_den)
    plt.plot(f, Pxx_den, marker='o', linestyle='none', color='red')
    plt.title("Identified Cycle of %i" % (cycle))
    plt.xlabel('frequency [Hz]')
    plt.ylabel('PSD [V**2/Hz]')
    plt.show()
    return( index, f, Pxx_den)
```

对原始数据使用这个函数，会出现图9.8所示的图像，很明显有一个周期为 12 个月的季节性。虽然考虑到这个数据的本质是长江水文资料，12 个月的周期是非常自然的预期，但是这个方法展示了对 ACF 序列运用周期图法（periodogram）找季节性周期的可靠性。从图9.8中可以看到，周期图法正确地识别出季节性的周期为 12 个月。因此我们需要将原始数据做 12 个月间隔的差分来消除季节性。然后对消除掉季节性的数据继续分析其平稳性、ACF 和 PACF 函数特性，从而确定最后所需的 ARIMA 模型结构。

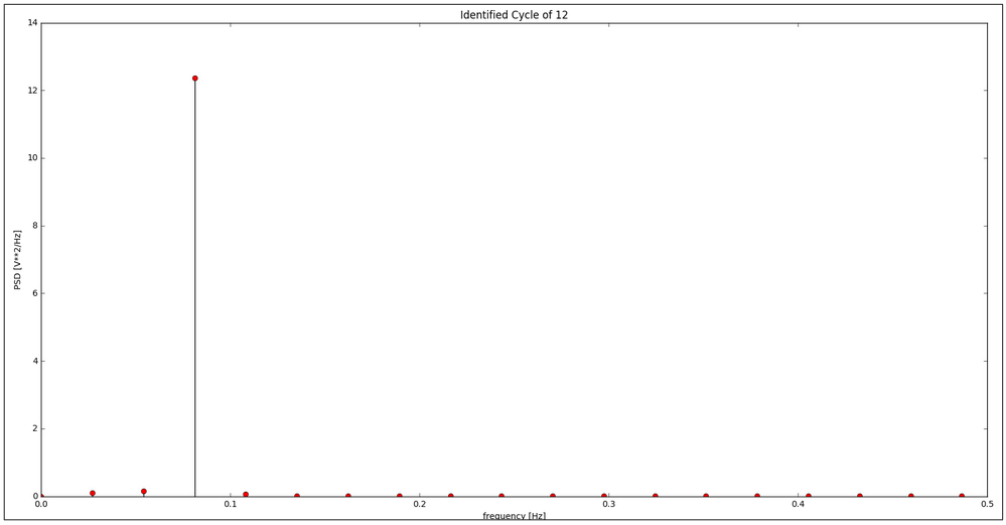


图 9.8　使用周期图法识别周期

```
Seasonality=12
waterFlowS12 = train.diff(Seasonality)[Seasonality:]
adftestS12 = sm.tsa.stattools.adfuller(waterFlowS12)
print("ADF test result shows test statistic is %f and p-value is %f" %(
adftestS12[:2]))

```

```
nlag=36
xvalues = np.arange(nlag+1)

acfS12, confiS12 = sm.tsa.stattools.acf(waterFlowS12, nlags=nlag, alpha
=0.05, fft=False)
confiS12 = confiS12 - confiS12.mean(1)[:,None]

fig = plt.figure()
ax0 = fig.add_subplot(221)
waterFlowS12.plot(ax=ax0)

ax1=fig.add_subplot(222)
sm.graphics.tsa.plot_acf(waterFlowS12, lags=nlag, ax=ax1)
plt.show()
```

如图9.9所示，去掉季节性的数据图像显示这个时间序列有非常强的均值回归行为，ACF 图显示为逐渐递减的序列直到第 12 个滞后项，但是 12 的倍数滞后项都没有显著的数据值，这说明两点：首先将数据去掉季节性是比较成功的，其次去掉季节性的数据仍然需要再做一阶差分。

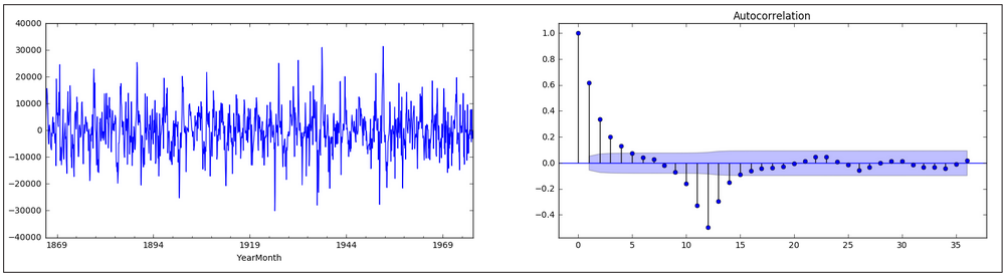


图 9.9　去掉季节性以后序列及其 ACF 图

去掉季节性的数据再做一阶差分以后就具备较好的统计特性，可以用于 Box-Jensen ARIMA 模型建模。我们现在已经知道去掉季节性的数据需要一个积分项，现在需要确定自回归和移动平均部分的滞后阶数。下面按照前面介绍的方法分析一阶差分以后的数据的 ACF 图和 PACF 图。其中，检视 PACF 图可以知道需要多少自回归滞后项，而 ACF 图可以告诉我们需要多少移动平均滞后项，如图9.10所示。

```
fig = plt.figure()
ax0 = fig.add_subplot(221)
sm.graphics.tsa.plot_acf(waterFlowS12d1, ax=ax0, lags=48)

```

```
ax1 = fig.add_subplot(222)
sm.graphics.tsa.plot_pacf(waterFlowS12d1, ax=ax1, lags=48)
plt.show()
```

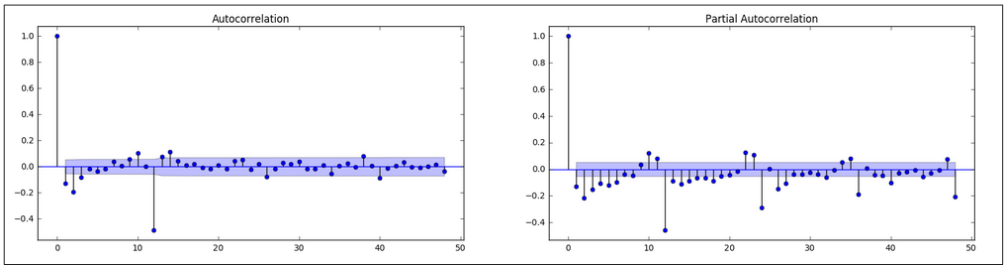


图 9.10　做了一阶差分和季节性差分之后的时间序列的 ACF 图和 PACF 图

根据前面提到的识别模型季节性的规则，ACF 图表明可能需要 1 个季节性预测误差滞后项，即差分滞后项。这表明我们需要一个季节性 ARIMA 模型，即 SARIMA(*p,d,q*)(*P,D,Q,S*) 模型。在 Python 里，可以用 StatsModels 中的状态空间模型来拟合 SARIMA 模型的参数。

```
mod1 = sm.tsa.statespace.SARIMAX(train, trend='n', order=(0,1,0),
seasonal_order=(0,1,1,12)).fit()
pred=mod1.predict()
print(mod1.summary())
```

模型拟合结果如9.11所示。

```
                                 Statespace Model Results
==========================================================================================
Dep. Variable:                           WaterFlow   No. Observations:                 1344
Model:             SARIMAX(0, 1, 0)x(0, 1, 1, 12)   Log Likelihood              -13198.909
Date:                            Tue, 03 Jan 2017   AIC                          26401.819
Time:                                    23:51:26   BIC                          26412.226
Sample:                                01-01-1865   HQIC                         26405.717
                                     - 12-01-1976
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ma.S.L12      -0.9474      0.009   -102.636      0.000      -0.965      -0.929
sigma2      2.368e+07   2.22e-11   1.07e+18      0.000    2.37e+07    2.37e+07
===================================================================================
Ljung-Box (Q):                      120.46   Jarque-Bera (JB):               284.24
Prob(Q):                              0.00   Prob(JB):                         0.00
Heteroskedasticity (H):               1.21   Skew:                             0.06
Prob(H) (two-sided):                  0.04   Kurtosis:                         5.26
===================================================================================
```

图 9.11　SARIMA(0,1,0)(0,1,1,12) 模型拟合结果

一般说来，这种季节性明显的数据，使用 SARIMA 模型拟合结果会比较好，特别是样本内数据。下面来看一看样本内数据的效果，包括模型检验的结果，如图9.12和图9.13所示。

```
subtrain = train['1960':'1970']
MAPE = (np.abs(train-pred)/train).mean()
subMAPE = (np.abs(subtrain-pred['1960':'1970'])/train).mean()

fig = plt.figure()
ax0 = fig.add_subplot(211)
plt.plot(pred, label='Fitted');
plt.plot(train, color='red', label='Original')
plt.legend(loc='best')
plt.title("SARIMA(0,1,0)(0,1,1,12) Model, MAPE = %.4f" % MAPE)

ax1 = fig.add_subplot(212)
plt.plot(pred['1960':'1970'], label='Fitted');
plt.plot(subtrain, color='red', label='Original')
plt.legend(loc='best')
plt.title("Details from 1960 to 1970, MAPE = %.4f" % subMAPE)
plt.show()
```

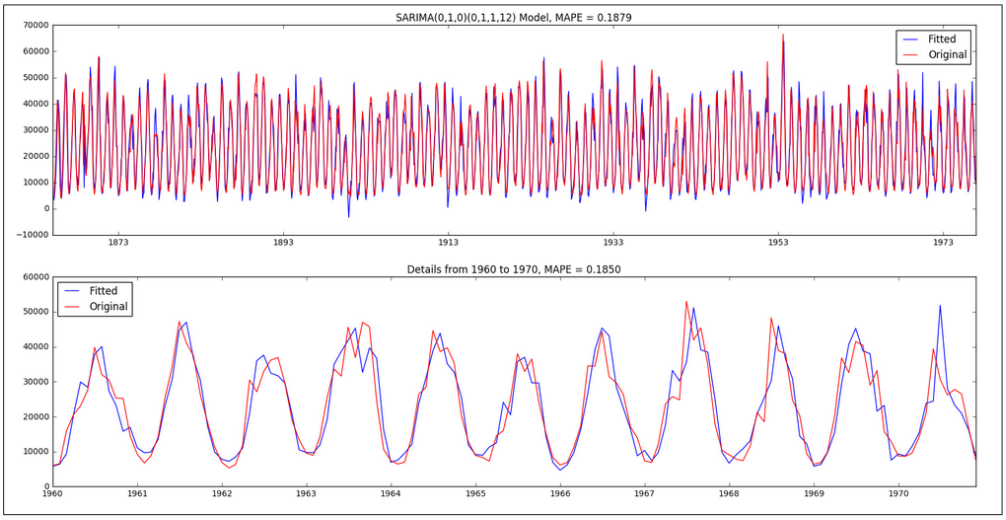


图 9.12 SARIMA 模型拟合结果

我们同时也测试了添加一个季节性自回归项，或者添加一个普通的自回归项/移动平均项的 SARIMA 模型，其结果都没有当前的模型好，MAPE 值分别上升到 19.1% 和 19.8%。只有当同时添加一个普通自回归项和一个普通移动平均项，即 SARIMA(1,1,1)(0,1,1,12) 模型，样本内数据的 MAPE 值降为 17.2%。我们使用这两个 MAPE 值低于 19% 的模型来进行样本外数据的预测。

```
forecast1 = mod1.predict(start = '1976-12-01', end='1978' , dynamic= True)
forecast2 = mod2.predict(start = '1976-12-01', end='1978' , dynamic= True)
MAPE1 = ((test-forecast1).abs() / test).mean()*100
MAPE2 = ((test-forecast2).abs() / test).mean()*100

plt.plot(test, color='black', label='Original')
plt.plot(forecast1, color='green', label='Model 1 : SARIMA(0,1,0)(0,1,1,12)'
)
plt.plot(forecast2, color='red', label='Model 2 : SARIMA(1,1,1)(0,1,1,12)')
plt.legend(loc='best')
plt.title('Model 1 MAPE=%.f%%; Model 2 MAPE=%.f%%'%(MAPE1, MAPE2))
```

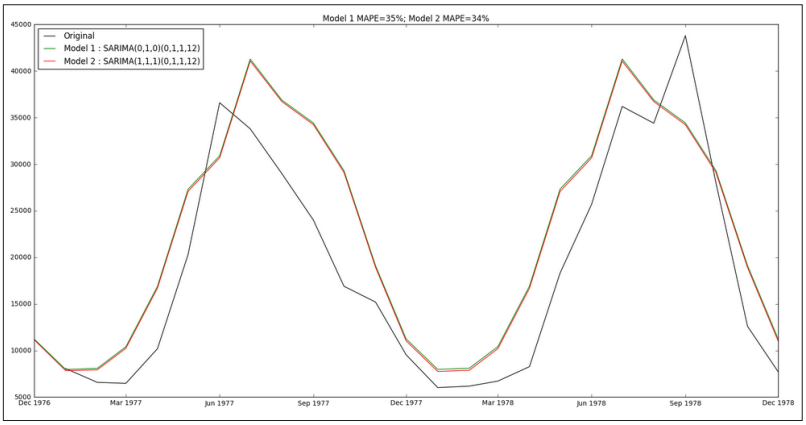


图 9.13　SARIMA 模型预测结果对比

我们看到这两个 SARIMA 模型在最后的预测上没有本质的区别，几乎重合，因此我们更倾向于保留比较简单、更稳定的 SARIMA(0,1,0)(0,1,1,12) 模型。

看了 SARIMA 模型的表现，下面再看一看深度学习能不能有更强的预测能力，以及如何构造这么一个预测能力更强的深度学习模型。这里选择构造一个 LSTM 模型，因为 LSTM 模型在绝大多数应用中属于 RNN 类模型里面表现最好的。用 Keras 构造 LSTM 模型只需要遵循以下几个简单的步骤即可，当然对于模型的理解必不可少。

(1) 将数据标准化，可以使用 z-score 法，也可以将数据纳入 [0,1] 区间。我们选择后者，因为更简单。

(2) 按照需要的神经网络模型构造数据格式。Keras 软件包的 LSTM 神经网络模型要求输入的自变量数据按照 [样本数，时间步，特征变量数] 的三维格式来组织，即我们应该按照每一个样本点对应一个时间步，一个时间步包含多个特征变量来构造，而因变量矩阵的维度则相应为 [样本数，前进时间步]。如何组织数据反映了所选择的神经

网络模型的结构。注意，如果时间步为 1，则 LSTM 模型与一个简单的前馈神经网络是一样的，也就是说这个模型变为一个非线性的自回归模型了。

(3) 按照 Keras 要求定义深度学习模型，比如对于时间序列模型一般就是要定义一个序列模型（Sequential），通常是多个网络层的线性堆叠，因此需要在这个模块中添加不同的神经网络层，一般是先加一个 LSTM 层作为输入信息和隐藏状态的桥梁，再加一个全连接层（Dense）来链接隐藏状态和输出信息。

```
def create_dataset(dataset, timestep=1, look_back=1, look_ahead=1):
    from statsmodels.tsa.tsatools import lagmat
    import numpy as np
    ds = dataset.reshape(-1, 1)
    dataX = lagmat(dataset, maxlag=look_back, trim="both", original='ex')
    dataY = lagmat(dataset[look_back:], maxlag=look_ahead, trim="backward",
    original='ex')
    # reshape and remove redundent rows
    dataX = dataX.reshape(dataX.shape[0], timestep, dataX.shape[1])[:-(
    look_ahead-1)]
    return np.array(dataX), np.array(dataY[:-(look_ahead-1)])
```

现在我们需要将原始数据分成建模数据部分和验证部分。不过我们在前面已经将数据提前划分了，保留了最后 24 个月的数据作为验证部分，而其余的则为建模训练集部分。下面我们需要将数据标准化一下，把取值范围纳入 [0,1] 区间，这样能提高计算的稳定性。另外，因为 Keras 要求输入数据为 numpy 多维数组形式，而不是 pandas 的数据框形式，因此我们还要转换一下数据的格式。

```
scaler = MinMaxScaler(feature_range=(0, 1))
trainstd = scaler.fit_transform(train.values.astype(float).reshape(-1, 1))
teststd = scaler.transform(test.values.astype(float).reshape(-1, 1))

lookback=60
lookahead=24
timestep=1
trainX, trainY = create_dataset(trainstd, timestep=1, look_back=lookback,
look_ahead=lookahead)
```

现在定义我们的 LSTM 模型。

```
batch_size=11
model = kModels.Sequential()
model.add(kLayers.LSTM(48, batch_size=batch_size, input_shape=(1, lookback),
 kernel_initializer='he_uniform')
model.add(kLayers.Dense(lookahead))
model.compile(loss='mean_squared_error', optimizer='adam')
```

下面开始对这个模型进行拟合，拟合迭代次数为 20 次，批量数（batch_size）为 1。一般说来，批量数越小，在其他参数不变的情况下拟合的效果越好，但是时间也越长，过度拟合的风险也越高。通过将参数 verbose 设为 0，要求不显示拟合过程中的输出状态。如果将这个参数设置为 1，则会显示最终的拟合结果，如果将这个参数设置为 2，则会将每次迭代的结果显示出来，如果是在批量处理（batch）模式下执行这个程序，则显示终端会显示一个字符形式的进度条。

```
model.fit(trainX, trainY, epochs=20, batch_size=batch_size, verbose=0)
```

如果使用 CPU，则拟合用时 6 分 6 秒，如果使用 GTX1060 GPU，则拟合用时只需要 30 秒左右，性能差别之大可见一斑。这里明显体现了 CNTK 计算后台的速度优势。如果采用 Theano 作为计算后台，在使用同样的 GPU 进行计算的情况下，一共耗时 1 分 23 秒。

有了模型以后，下面准备处理测试数据进行预测。前面保留了最后 24 个月的数据作为测试数据，而我们的 LSTM 模型回看的时间是 48 个时间点，因此不能直接将最后 24 个月的测试数据使用 create_dataset 来生成供预测函数使用的数据。但是因为这个模型一次性往前预测 12 期，因此可以直接抓取待预测的 24 个月之前的 48 个月数据，将其标准化以后使用 reshape 函数变为合乎要求的格式并带入预测函数进行预测，再与实际测试数据和 SARIMA 模型进行比较。如果要像 SARIMA 模型一样往前预测 24 个月，那么直接带入已经预测好的 12 个月数据作为新的输入继续预测后面 12 个月，这与 SARIMA 模型的“动态”（dynamic=True）是一样的。另外，使用 Keras 的模型，可以选择多种损失函数（Loss Function）作为优化标准。对于时间序列，因为最后比较的是 MAPE 值，因此在 Keras 中我们选择使用 loss='mape' 作为参数。

首先预测测试数据中的前半部分，即 1977 年的月度水流量数据。输入数据为前 48 个月，即 1973 年到 1976 年的月度水流量，变换为 [1, 1, 60] 的维度以后输入预测函数，得到 1977 年的月度水流量预测数据。执行下面的代码就可以进行相应的预测和 MAPE 的计算并绘图，得到如图9.14所示的结果。

```
feedData = scaler.transform(df1.WaterFlow['1972':'1976'].reshape(-1, 1)).
copy()
feedX = (feedData).reshape(1, 1, lookback)
feedX = (feedX)
prediction1 = model.predict(feedX)

predictionRaw = scaler.inverse_transform(prediction1.reshape(-1, 1))
actual1 = df1.WaterFlow['1977':'1978'].copy().reshape(-1, 1)
MAPE = (np.abs(predictionRaw-actual1)/actual1).mean()

plt.plot(predictionRaw, label='Prediction')
plt.plot(actual1, label='Actual')
plt.title("MAPE = %.4f" % MAPE)
plt.legend(loc='best')
plt.xlim((0, 23))
plt.xlabel("Month")
plt.show()
```

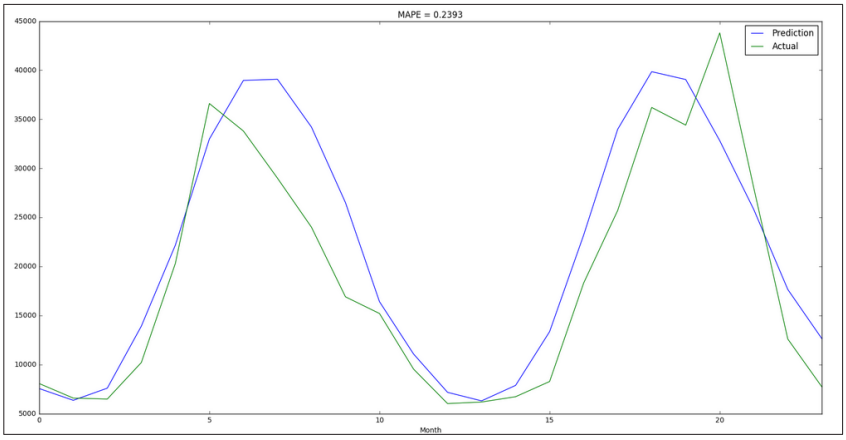


图 9.14 LSTM 模型预测结果和实际测试数据比较

我们看到，预测的 MAPE 值只有 24% 左右，比 SARMIMA 模型的 35% 要低很多，尤其是拟合的曲线更加平滑。上面的试验显示，一个简单的 LSTM 模型在拟合这种周期性很强的时间序列上具备较好的性能，尤其是不需要建模统计师具体分析周期的多少，然后再对自回归项和移动平均项参数进行选择，从而大大降低了建模的难度。分析师可以直接对几个关键参数进行逐一筛选，用计算机取代人力，从而可以有效提高工作效率。

前面提到，LSTM 只是 Sequential 模型的一个层，而在 Sequential 模型里面可以叠加多个 LSTM 模型层，构造一个深度 RNN 模型，这一点类似使用 MLP 来提升传统前置神经网络的预测能力。下面介绍如何构造一个叠加 LSTM 的序列模型，希望不同的 LSTM 层能捕捉不同的波动。当然，这种类型的模型更加复杂，更容易过度拟合，在数据比较简单的情况下不一定比前面的单层 LSTM 加一个 Dense 层更有效，如图9.14所示。

下面试一试一个叠加的 LSTM 模型。在这个模型里，我们叠加了两个 LSTM 层，每层都使用了 10% 的 Dropout 来防止过度拟合，最后加一个全连接层再输出结果。注意，我们在第一层的 LSTM 层里制定了输入数据的维度，并把 return_sequences 参数设置为 True。这个参数指定是将上一次计算的输出返回到下一层数据，还是将原始的整个序列返回到下一层数据。如果将 return_sequences 参数指定为 True，则将原始的整个序列返回。这就是说，在多个叠加的 LSTM 层里面，我们需要返回原始的序列数据而不是计算出来的序列数据供下一层使用。同时，在第二层里面，我们无须再指定输入数据的维度，因为 Keras 能自动分析出这个参数。另外，因为 return_sequences 参数的默认值为 False，因此不许要特别设置，即我们在叠加的最后一个 LSTM 层里不返回原始的序列数据，而是返回计算出的序列数据供输出层或者全连接层使用。

```
%%time
# create and fit the Stacked LSTM network
batch_size=1
model2 = kModels.Sequential()
model2.add(kLayers.LSTM(96, batch_size=batchsize, input_shape=(1, lookback),
 return_sequences=True))
model2.add(kLayers.Dropout(0.1))
model2.add(kLayers.LSTM(48)))
model2.add(kLayers.Dense(lookahead))
model2.compile(loss='mape', optimizer='adam')
model2.fit(trainX, trainY, epochs=15, batch_size=batch_size, verbose=0)
```

使用 GPU 拟合这个叠加的模型耗时大约 45 秒。那么结果如何呢？我们执行下面的程序来展示预测和实际的曲线。

```
feedData = df1.WaterFlow['1972':'1976'].copy()
feedX = scaler.transform(feedData.reshape(-1, 1)).reshape(1, 1, lookback)
prediction2 = model2.predict(feedX)
predictionRaw = scaler.inverse_transform(prediction2.reshape(-1, 1))
actual1 = df1.WaterFlow['1977':'1978'].copy().reshape(-1, 1)
MAPE = (np.abs(predictionRaw-actual1)/actual1).mean()
```

```
plt.plot(predictionRaw, label='Prediction')
plt.plot(actual1, label='Actual')
plt.title("MAPE = %.4f" % MAPE)
plt.legend(loc='best')
plt.xlim((0, 23))
plt.xlabel("Month")
plt.show()
```

从图9.15中可以看，预测的效果与简单的单层 LSTM 的模型差不多，MAPE 略微低于 24%，并没有显著的差异。造成这种结果的原因可能有两种，一种可能是这个模型迭代次数过多，有过度拟合的情况，另一种可能是需要更多 LSTM 层来抽取更为细节的信息。为了验证这两种假设，我们下面编写一个函数，可以控制多个 LSTM 层的叠加数量，也可以控制迭代的次数和批量数的大小。然后我们选择在第一层 LSTM 之外额外添加两层 LSTM，每层都有 10% 的 Dropout 比率来控制过度拟合，最后用一个全连接层连接输出层。我们选择不同的迭代次数，从 4 次到 10 次，看看最后结果如何，并且输出每次迭代的运算时间。

```
def SLSTM(epoch=10, stacks=1, batchsize=5):
    batch_size=batchsize
    model2 = kModels.Sequential()
    model2.add(kLayers.LSTM(48, batch_size=batchsize, input_shape=(1,
    lookback), return_sequences=True))
    model2.add(kLayers.Dropout(0.1))
    for i in range(stacks-1):
        model2.add(kLayers.LSTM(32, return_sequences=True))
        model2.add(kLayers.Dropout(0.1))
    model2.add(kLayers.LSTM(32, return_sequences=False))
    model2.add(kLayers.Dense(lookahead))
    model2.compile(loss='mape', optimizer='adam')
    t0 = time()
    model2.fit(trainX, trainY, epochs=epoch, batch_size=batch_size, verbose
    =0)

    feedData = df1.WaterFlow['1972':'1976'].copy()
    feedX = scaler.transform(feedData.reshape(-1, 1)).reshape(1, 1, lookback
    )
    prediction2 = model2.predict(feedX)
```

```
    predictionRaw = scaler.inverse_transform(prediction2.reshape(-1, 1))
    actual1 = df1.WaterFlow['1977':'1978'].copy().reshape(-1, 1)
    deltatime = time()-t0
    MAPE = (np.abs(predictionRaw-actual1)/actual1).mean()
    print("Epoch= %.1f, MAPE=%.5f, 消耗时间=%.4f 秒" % (epoch, MAPE,
    deltatime))

for epoch in [4,5,6,7,8,9,10]:
    SLSTM(epoch, stacks=2, batchsize=5)
```

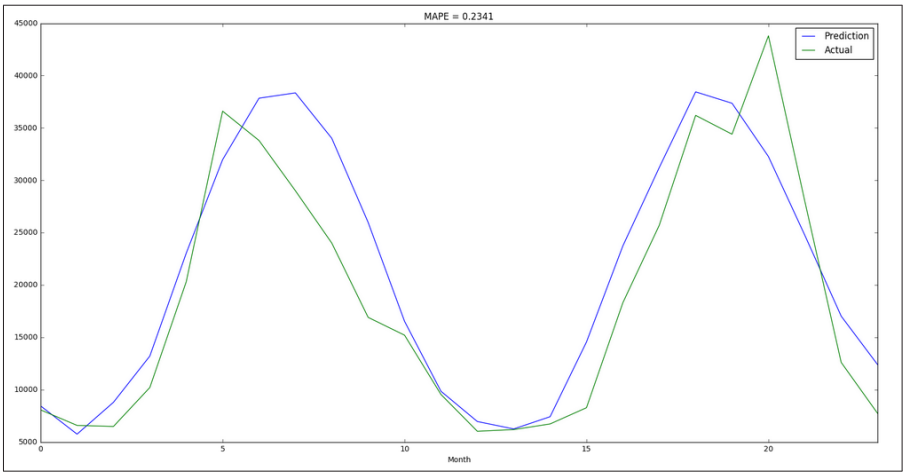


图 9.15　叠加的 LSTM 模型预测结果和实际测试数据比较

表9.1展示了每次计算的结果和时间。可以看到，首先，每增加一次迭代，计算时间大约增加 5 秒钟，同时，在 6 次迭代的时候就可以达到低于 25% 的 MAPE，然后随着迭代次数的增加误差指标回升到 26% 以上，最后在第 9 次迭代的时候误差降到 22% 以下。这说明通过增加 LSTM 模型叠加的层数，确实可以让模型更快地学习数据的模式，但是在使用叠加层数的 LSTM 模型时，需要适当降低迭代的次数，并通过 Dropout 等技术来防止过度拟合。

表 9.1　叠加 LSTM 模型训练结果

Epoch= 4.0,	MAPE=0.28090,	消耗时间 =41.0717 秒
Epoch= 5.0,	MAPE=0.25532,	消耗时间 =51.6239 秒
Epoch= 6.0,	MAPE=0.24873,	消耗时间 =61.5472 秒
Epoch= 7.0,	MAPE=0.25706,	消耗时间 =72.0120 秒
Epoch= 8.0,	MAPE=0.26800,	消耗时间 =81.7846 秒
Epoch= 9.0,	MAPE=0.21710,	消耗时间 =92.5837 秒
Epoch= 10.0,	MAPE=0.25783,	消耗时间 =102.8702 秒

9.7.2 国际航空月度乘客数时间序列模型

下面使用前面学到的技术来对国际航空月度乘客数进行建模和预测。具体来讲，我们先用传统的季节性自回归积分移动平均模型（SARIMA）来拟合和预测这个时间序列，然后再分别对不进行平稳化和进行平稳化以后的数据进行拟合，展示 LSTM 模型在两种情况下的表现。在使用传统的 SARIMA 模型进行拟合之前，我们仍然要对该数据的平稳性进行分析，如果该数据不是平稳的，那么要进行适当的差分操作使其变为平稳的序列，同时使用周期图法侦测季节性的长度。这些基本的数据分析完成以后，得到所需的参数，然后拟合模型，预测最后 21 个月的序列数据，考察误差大小（因为数据截止到 1960 年 9 月）。注意，在用于模型拟合的训练数据中是不包含最后这部分的测试数据的。

虽然这个数据明显不是平稳的，但是我们依然按照标准的步骤先来检验数据的平稳性，结果如图9.16所示。

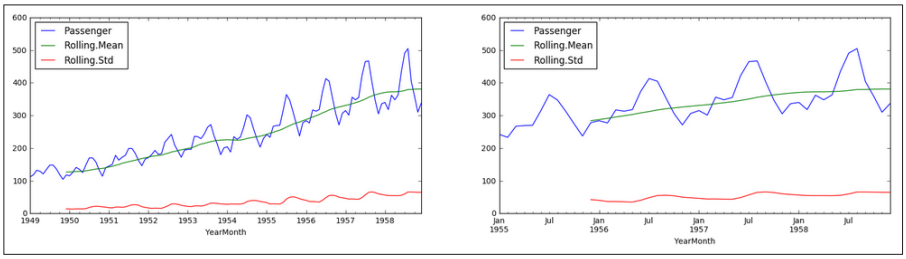


图 9.16　月度乘客数量序列平稳性检测

```
cutoff=21
train2 = df2.Passenger[:-cutoff]
test2 = df2.Passenger[-cutoff:]

fig = plt.figure()
ax0 = fig.add_subplot(221)
adftest, dftest0=test_stationarity(train2)
dftest0.plot(ax=ax0)
print('原始数据平稳性检验')
print(adftest)

ax1 = fig.add_subplot(222)
adftest, dftest1=test_stationarity(train2['1955':'1960'])
dftest1.plot(ax=ax1)
print('局部数据平稳性检验')
```

当然，这个数据具有明显的季节性周期，并且用肉眼可以识别为 12 个月，但是很多时候我们都需要一种自动化建模的手段，因此下面仍然使用周期图法来侦测周期性。图9.17的结果表明，在数据不平稳的情况下，侦测效果不好。如图9.17所示的这个数据的季节性周期明显在 12 个月左右，但是侦测结果显示在 37 个月，扣除误差，也是实际季节性周期的 3 倍左右。这表明要使用周期图法来侦测周期性，至少要求数据没有明显的趋势性。

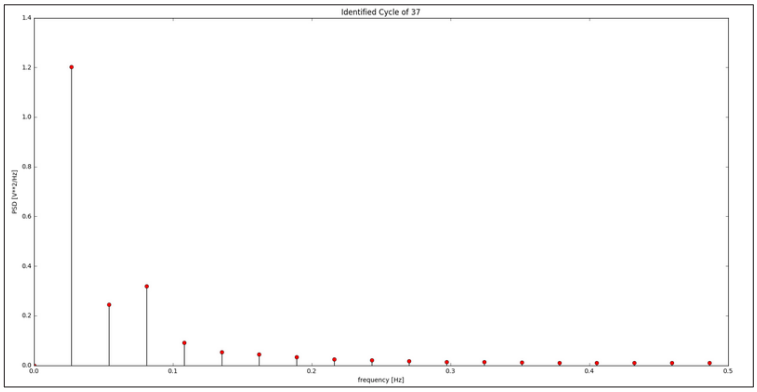


图 9.17　原始数据使用周期图法侦测周期性

```
index, f, Power = CalculateCycle(train2)
```

那么先进行一阶差分将趋势性去掉，即使差分过后的数据仍然存在异方差性，周期图法的方法也能比较容易发现数据的周期为 12，由图9.18 可以看出来。目前看来，异方差性如果是单调增减的，那么对数据的周期性侦测影响不大。

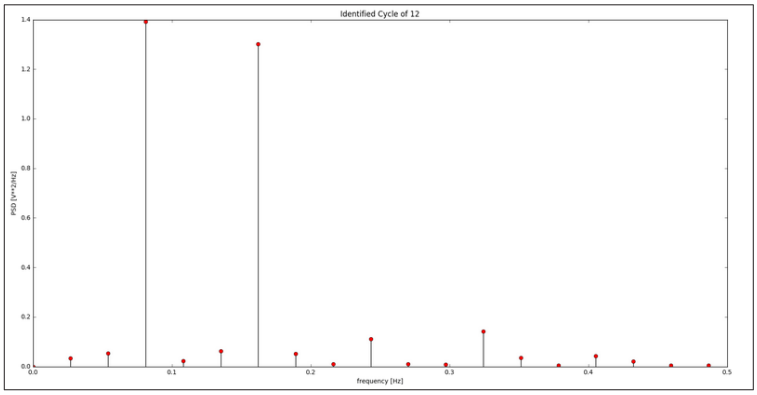


图 9.18　一阶差分后的数据使用周期图法方法侦测周期性

```
train2d1 = train2.diff(1)[1:]
```

```
index, f, Power = CalculateCycle(train2d1)
```

知道季节性周期的长度以后，我们做季节性差分就可以消除影响，使数据更为接近平稳的序列。对于一阶差分并去掉季节性的数据运用 ACF 和 PACF 来检查。

```
fig = plt.figure()
ax0 = fig.add_subplot(221)
sm.graphics.tsa.plot_acf(train2d1s12, ax=ax0, lags=48)

ax1 = fig.add_subplot(222)
sm.graphics.tsa.plot_pacf(train2d1s12, ax=ax1, lags=48)
plt.show()
```

图9.19 显示其不具备任何显著的自回归或者移动平均项，或者最多使用一个一阶自回归项即可。因此对于 SARIMA 模型的参数，可以设定为季节性部分参数为（1,1,0,12），对于非季节性部分的参数可以相应设置为（0,1,0）。

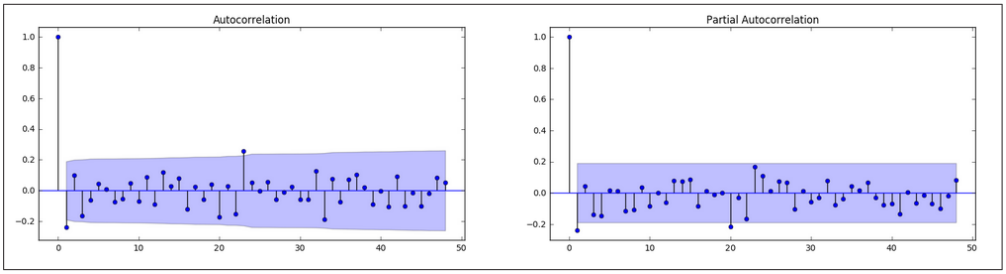


图 9.19　消除季节性和趋势性的数据的 ACF 图和 PACF 图

```
mod1 = sm.tsa.statespace.SARIMAX(train2, trend='c', order=(0,1,0),
seasonal_order=(1,1,0,12)).fit()
pred=mod1.predict()
print(mod1.summary())
```

拟合的结果见图9.20。

这个拟合的结果是比较好的，对于训练集数据，其综合 MAPE 仅仅为 5% 左右，而局部 MAPE 值，比如 1955 年到 1956 年这 24 个月的数据对应的 MAPE 值甚至只有 2.7%，如图9.21所示。

当然模型的预测效果最后还是要看在测试数据上的表现。1959 年 1 月到 1960 年 9 月的测试数据与预测数据的对比表明，MAPE 值为 12.4%，显示 SARIMA 模型有一定过度拟合的可能。从图9.22可以看出，误差主要原因是预测的数据不能有效地拟合乘客

数量不停增长的趋势，1960 年的预测数据和 1959 年的预测数据的水平大致一样，但是实际数据是 1960 年比 1959 年平均乘客数提高了超过 10%。但是这个趋势对于这个数据而言其实表现为异方差性随时间而逐渐增大，因为一阶差分以后的数据总平均值保持在一个接近零的水平，并且不随时间而变动，只有数据的波动随时间增大。

```
                                Statespace Model Results
==========================================================================================
Dep. Variable:                           Passenger   No. Observations:                  120
Model:             SARIMAX(0, 1, 0)x(1, 1, 0, 12)   Log Likelihood                -402.338
Date:                            Sun, 12 Feb 2017   AIC                            810.677
Time:                                    02:30:15   BIC                            819.039
Sample:                                01-01-1949   HQIC                           814.073
                                     - 12-01-1958
Covariance Type:                              opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
intercept     -0.0204      1.026     -0.020      0.984      -2.031       1.990
ar.S.L12      -0.1020      0.085     -1.194      0.232      -0.269       0.065
sigma2       107.9052     12.268      8.796      0.000      83.861     131.949
===================================================================================
Ljung-Box (Q):                       51.88   Jarque-Bera (JB):                 4.66
Prob(Q):                              0.10   Prob(JB):                         0.10
Heteroskedasticity (H):               1.38   Skew:                            -0.18
Prob(H) (two-sided):                  0.34   Kurtosis:                         3.96
===================================================================================

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).
```

图 9.20　消除季节性和趋势性的数据的 ACF 图和 PACF 图

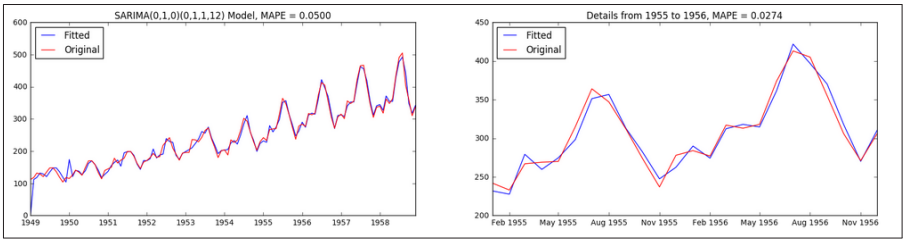


图 9.21　拟合结果

看了 SARIMA 模型在一个去掉趋势性但是仍然保留异方差性的序列上的表现后，再看一看循环神经网络在原始数据上的表现。即构造一个简单的单层 LSTM 模型，将其应用于不去掉趋势性、不消除异方差性的原始数据上，看一下神经网络模型能不能自动抓取数据的模式。

```
scaler = MinMaxScaler(feature_range=(0, 1))
trainstd2 = scaler.fit_transform(train2.values.astype(float).reshape(-1, 1))
teststd2 = scaler.transform(test2.values.astype(float).reshape(-1, 1))

```

```
lookback=60
lookahead=24
timestep=1
trainX2, trainY2 = create_dataset(trainstd2, timestep=1, look_back=lookback,
 look_ahead=lookahead)
```

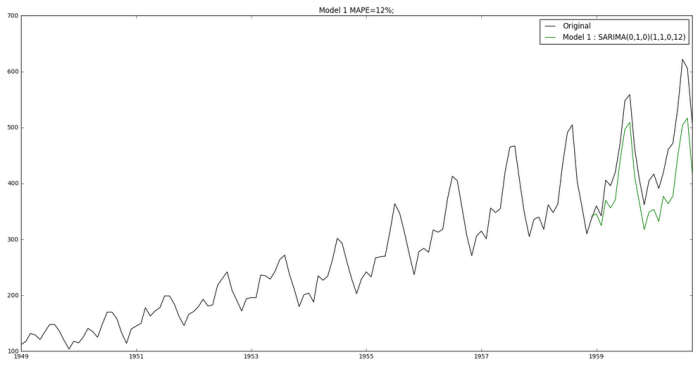


图 9.22 SARIMA(0,1,0)(1,1,0,12) 模型在测试数据上的预测效果

下面的代码构造了我们的 LSTM 神经网络模型，并进行拟合。因为数据量较少，拟合时间只用了不到 11 秒。

```
%%time
# create and fit the simplest LSTM network
batch_size=1
model = kModels.Sequential()
model.add(kLayers.LSTM(96, batch_size=batch_size, input_shape=(1, lookback),
 kernel_initializer='he_uniform'))
#model.add(kLayers.Dense(32))
model.add(kLayers.Dense(lookahead))
#model.compile(loss='mean_squared_error', optimizer='adam')
model.compile(loss='mape', optimizer='adam')
model.fit(trainX2, trainY2,  epochs=30, batch_size=batch_size, verbose=0)
```

拟合完毕以后就可以对测试数据进行预测，并检验平均误差百分比，如图9.23 所示。我们注意到有几处和 SARIMA 模型预测能力不同的地方。

(1) 首先，其 MAPE 值只有 11.6%，比 SARIMA 模型的预测稍微低一点点，但是并没有显著的区别，但是这个比较容易调整。当把迭代次数提高到 40 次的时候，测试集数据的 MAPE 值可降低到 7.0%。

(2) 其次，LSTM 模型发现了逐渐增加的平均乘客数和增加的波动两个趋势，即使我们并没有在模型里指明或者对数据进行任何特别的处理；而 SARIMA 模型的预测并没有抓住乘客数波动范围增加的数据模式，即对异方差性不能自动解决，而是需要我们对数据进行一些变换，比如取对数等，来消减异方差性，从而得到乘客数波动范围增加这个特性，并在预测中反映出来。这样可以大大提高建模效率。

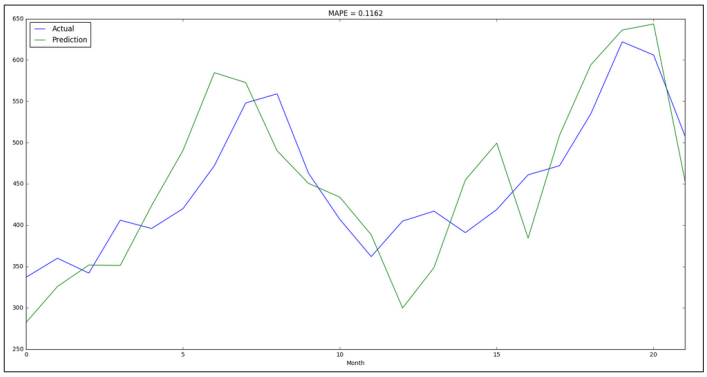


图 9.23　单层 LSTM 模型在测试数据上的预测效果

下面对原始数据取对数以尽量消除异方差性，然后再看一看 SARIMA 模型和单层 LSTM 模型各自的预测效果。首先根据图9.24，可以看出取对数以后的差分数据平稳性很好，12 个月的周期在 ACF 图和 PACF 图中都明显显示出来，而且都表现出更为明显的二阶自回归和一阶移动平均项要求。因此我们的 SARIMA 模型在季节性参数上使用 (2,1,1,12) 而不是原来的 (1,1,0,12)。预测数据在重新返回到原始的尺度上后，与实际数据比较的 MAPE 值现在降低到仅为 5%，如图9.25所示，相对于不消除异方差性的情况准确度提高了一倍，效果非常明显。并且现在的预测数据有效地体现乘客数量平均水平和波动幅度随时间增大的情况。

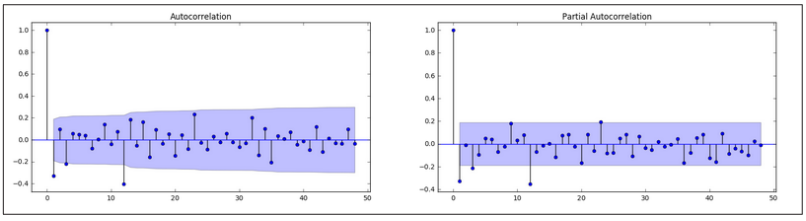


图 9.24　消除趋势和异方差性后的 ACF 图和 PACF 图

另一方面，对原始数据进行非线性处理之后，深度学习模型的预测能力退化较大，这是一个非常有趣的现象。无论是单层还是多层的 LSTM 模型，在数据的预测上都出现较大的偏差。这显示在时间序列建模中如果要使用深度学习的算法，则保持原有数据的相对比例很重要。

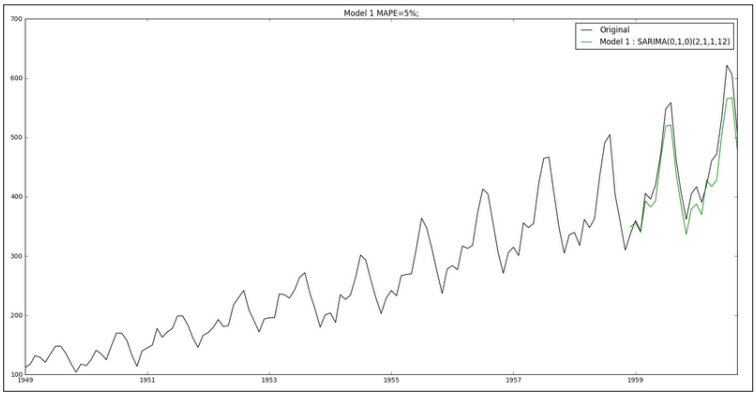


图 9.25 SARIMA(0,1,0)(1,1,1,12) 模型在平稳数据上的预测效果

9.8 总结

本章介绍了时间序列模型的基本概念，并介绍和比较了传统的 ARIMA 模型与新兴的 LSTM 深度学习模型。我们发现，当使用有一定周期性的时间序列数据进行建模和预测时，简单的 LSTM 模型已经能很容易地达到需要仔细分析周期性和各项建模参数的 (S)ARIMA 模型的预测效果。我们在这种情况下不需要花费大量的精力对数据进行处理来达到平稳性以便适用传统的 ARIMA 模型建模。相反，深度学习模型能够自动发现非平稳数据里的趋势性和异方差性，并对这两种数据的模式建模，在迭代一定的次数之后达到较好的预测效果。我们甚至能够叠加多个 LSTM 层，在迭代较少的次数时就能识别数据的特征进行预测。

当然，深度学习模型的算法属于随机算法，我们有时候需要进行多次拟合，试着设定不同的初始值才能得到合适的结果。深度学习模型也比较容易过度拟合，在不同层之间加入连接断开层（Dropout）或者在 LSTM 层设置递归权重或者偏置权重进行正则化处理，可以在一定程度上防止发生过度拟合。

另外，虽然使用一些非线性变换，比如取自然对数，能有效处理数据的异方差性，并帮助传统的 ARIMA 模型提高预测能力，但是这个方法对深度学习模型并不适用。当应用了变换后的数据来训练深度学习模型，其预测能力反而不如使用原始数据的时候效果好。

10 智能物联网

10.1 Azure 和 IoT

本章会介绍 IoT 解决方案架构，这个架构用 Azure 服务部署，而且包含 IoT 方案的常用特点。IoT 解决方案架构要求在设备之间存在安全的、双向的通信和一个后台的解决方案。后台解决方案可以用自动化的预测分析来揭示设备到云端的时间流内涵。Azure IoT Hub 是实现 IoT 方案架构的关键基石。IoT 套件为这个架构提供了完整的端到端的实现，从数据的收集、存储和整理到利用 AzureML 的机器学习和深度学习功能进行预测和分析，都可以在一个框架内完成。例如：

- 远程监控方案可以监控每个终端设备，比如贩卖机。
- 预测维护方案可以预期设备的维护需要而避免不必要的宕机，比如水泵站的水泵。

IoT 方案架构

图10.1展示了一个典型的 IoT 解决方案架构，它描述了其中关键的组成部分。在这个架构中，IoT 设备收集数据，发送到云端网关。云端网关让其他后端服务处理数据，处理完以后通过仪表板或者展示设备送到其他商业服务程序或者人工操作员。

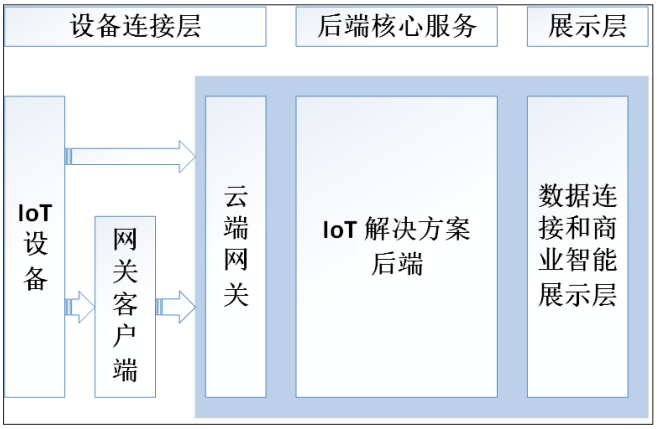


图 10.1 典型的 IoT 方案架构

设备连接

在这个 IoT 解决方案架构里，设备发送测量数据到云端服务并存储，然后处理，比如水泵站的传感器读数。在预测维护场景里，解决方案后端用到了传感器流数据，确定一根特定的水泵管子需要维护。设备还可以接受和回复来自云端的消息。比如，在预测维护场景里，解决方案后端可以发送消息到水泵站的正常水泵，在需要维修的水泵开始维修之前，把水流引导到那些正常的水泵中，这样维护工程师到达现场就可以立刻开始作业。在这个 IoT 项目里，一个最大的挑战是如何可靠、安全地连接设备和方案后台服务。IoT 设备和其他客户端 (比如浏览器，手机 APP) 比起来有以下不同的特征。

(1) IoT 设备往往是嵌入式设备，没有人类操作员。

(2) IoT 设备可以部署在遥远的地点，物理访问成本很高。

(3) IoT 设备仅可以通过方案后台访问，没有其他途径和设备交互。

(4) IoT 设备只拥有有限的电源和计算资源。

(5) IoT 设备可能会有断断续续的、缓慢的或者昂贵的网络连接。

(6) IoT 设备需要使用专门的或者行业特有的通信协议。

(7) IoT 设备可以被很多的通用硬件和软件平台制造出来。

除了以上要求，任何 IoT 解决方案必须具备扩展性、安全性和可靠性。连接要求导致用传统技术（比如网络容器或者消息代理）实现既难又耗时。Azure IoT Hub 和 Azure IoT 设备 SDK 使得实现这样的要求容易很多，一个设备可以直接和云端网关端点连接，如果设备无法使用任何云端网关支持的通信协议，则它可以通过中转网关连接。比如，如果设备不能使用任何 IoT Hub 支持的协议，则 Azure IoT 协议网关可以实现协议中转。

数据处理和分析

在云端，大部分数据处理运行在 IoT 解决方案后端，比如过滤测量数据和聚合测量数据，然后转发到其他服务。在 IoT 后台可以进行以下操作。

(1) 从设备处接收大规模测量数据，然后决定如何处理和存储这些数据。

(2) 可以从云端发送命令到一个特定的设备。

(3) 提供设备注册功能，这样可以开启和控制哪些设备可以连接到基础架构。

(4) 可以记录设备状态，监控设备行为。

在预测维护场景里，在方案后端存储了历史测量数据。方案后端可以用这些数据找出水泵何时需要维护的规律。IoT 解决方案可以包括自动反馈回路。例如一个分析模块可以从测量数据里识别一个特定设备的温度是否高于正常水平，然后方案后端可以发送命令给这个设备纠正偏差。

报表和商业连接

报表和商业连接层可以让终端用户连接 IoT 后端和设备，这样终端用户可以查看、分析设备传送的测量数据。这些视图可以以仪表板或者商业智能报表的形式展示历史数据和实时数据，例如，一名人工操作员可以查看特定水泵站的状态和任何系统报警。这一层服务整合了 IoT 解决方案和已有的商业应用并连接到企业业务流程和工作流程。例如，预测维护方案可以和调度系统整合，这样当水泵站需要检修的时候，调度系统可以调工程师检修这个水泵站。

下一步

Azure IoT Hub 提供了 IoT 解决方案和成千上万台设备之间的安全可靠的双向通信。有了 IoT Hub，方案后台具有以下功能。

(1) 从很多设备并发地接受测量数据。

(2) 把数据从设备导到事件流处理器。

(3) 上传文件到设备。

(4) 发送消息到特定的设备。

IoT Hub 还可以用于自己的 IoT 解决方案。而且 IoT Hub 提供了设备身份注册服务：可以激活设备、存储设备的安全证书和定义设备的访问 IoT Hub 的权限。

下面着重介绍 IoT Hub。

10.2　Azure IoT Hub 服务

本节会介绍为什么需要使用 IoT Hub 这个服务来实现 IoT 解决方案。Azure IoT Hub 是一个完全托管服务，其架构如图10.2所示。

(1) 提供了多个设备到云端和云端到设备的通信选项，包括单向消息、文件传输、请求–应答的方法。

(2) 提供内置的声明消息路由到其他 Azure 服务。

(3) 为设备提供了一个可查询存储元数据和同步状态信息。

(4) 安全通信和访问控制使用每个设备特定的安全密钥或 X.509 证书。

(5) 为设备连接和设备身份管理事件提供广泛的检测功能。

(6) 设备库支持受欢迎的语言和平台。

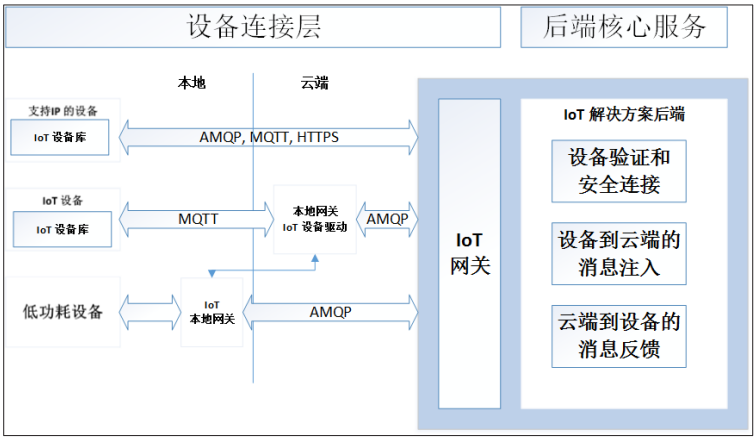


图 10.2　Azure IoT Hub 架构

为什么用 Azure IoT Hub

除了支持消息传递、文件算出和请求应答，IoT Hub 还解决了设备连接问题。

(1) 设备双系统。使用双系统，可以存储、同步和查询设备元数据和状态信息。设备副本其实是一个 JSON 文档，它存储了设备状态信息 (元数据、配置和条件) 。IoT Hub 为每个设备保留了持久的副本。

(2) 每个设备有特定认证和安全连接。可以为每个设备配置自己的安全密钥，使其能够连接到 IoT Hub。IoT Hub 标识注册表在设备解决方案中存储设备标识和密钥。其解决方案后端可以添加单个设备到允许或拒绝列表，从而实现对设备访问的完全控制。

(3) 基于声明性规则将设备到云消息路由到 Azure 服务。IoT Hub 允许我们根据路由规则定义消息路由，控制 Hub 发送设备到云消息的目的地。路由规则不需要我们编写任何代码，并且可以代替自定义 post-ingestion 调度程序。

(4) 监视设备连接操作。可以接收有关设备身份管理操作和设备连接事件的详细操作日志。此监视功能使我们的 IoT 解决方案能够识别连接设备问题，例如尝试使用错误凭据连接，过于频繁地发送消息或拒绝所有云到设备消息。

(5) 一组广泛的设备库。Azure IoT 提供了可用于各种语言和平台的 SDK——例如，多种 Linux 发行版、Windows 操作系统以及多种实时操作系统。Azure IoT 设备 SDK 还支持托管语言，如 C#、Java 和 JavaScript。

(6) IoT 协议和可扩展性。如果我们的解决方案无法使用设备库，则 IoT Hub 会公开一个公开协议，使设备能够在本机上使用 MQTT v3.1.1、HTTP 1.1 或 AMQP 1.0 协议。还可以扩展 IoT Hub，通过以下方式提供对自定义协议的支持。

- 使用 Azure IoT 网关 SDK 创建字段网关，将我们的自定义协议转换为 IoT Hub 服务支持的三种协议之一。
- 自定义 Azure IoT 协议网关，这是一个运行在云端的开源软件。
- Azure IoT 中心扩展到数百万个同时连接的设备和每秒数百万的事件。

网关

IoT 解决方案中的网关有两种形式：部署在云端的协议网关，以及部署在本地的本地网关。协议网关的工作是协议转换，比如从 MQTT 转换到 AMQP。本地网关在本地分析数据，做出实时决定，这样降低了来去云端的延迟和泄露隐私的风险。这两种网关都扮演了设备和 IoT Hub 之间的媒介。解决方案可以同时包括协议网关和本地网关。

IoT Hub 如何工作

Azure IoT Hub 通过实现了 Service-Assisted 通信模式来连接设备和方案后端。Service-Assisted 通信模式的目标是在一个控制系统和一个特殊目的设备之间建立互信，双向通信。这个通信模式具有以下原则。

(1) 安全第一，优先于其他所有功能。

(2) 设备不主动接收命令。如果设备想接收命令，则设备必须定时地和后端建立连接，查看是否有需要执行的命令。

(3) 设备应该只连接或建立与它们对等的已知服务 (比如 IoT Hub) 的路由。

(4) 设备和服务之间或者设备和网关之间的通信路径要在应用协议层保证是安全的。

(5) 系统级授权和身份验证基于每个设备的身份，使得访问凭证和权限几乎立刻撤销。

(6) 保持命令和设备消息直到设备连接收到它们，这样有助于电源或者连接不稳定的设备双向通信。IoT Hub 为每个设备维护命令队列。

(7) 应用程序有效载荷数据被单独保护，用于网关到特定服务的受保护传输。

移动行业已经大规模使用 Service-Assisted 通信模式来实现推送通知服务，比如 Windows 推送通知服务、Google 云消息、Apple 推送通知服务。

下一步

下面介绍 IoT Hub 如何实现基于标准的设备管理，以便远程管理配置和更新设备。

10.3 使用 IoT Hub 管理设备概述

介绍

Azure IoT Hub 提供了功能和可扩展性模型，使得设备和后端开发人员能够建立强大的设备管理解决方案。设备范围从受限传感器到单用途微控制器，再到强大的网关 (为设备群组通信路由)。此外，IoT 运营商的应用场景和要求在不同行业中非常不同。尽管存在着变化，使用 IoT Hub 设备管理提供的功能、模式和代码库，可以满足各种设备和最终用户的需求。创建成功的企业 IoT 解决方案的关键部分是为运营商如何处理他们的设备集合的持续管理提供战略。IoT 运营商需要见到可靠的工具和应用程序，使他们能够专注更具战略性的方面。本节主要包括以下内容。

(1) Azure IoT Hub 对设备管理方法的简要概述。

(2) 常见设备管理原则介绍。

(3) 设备生命周期介绍。

(4) 常见设备管理模式概述。

设备管理原则

IoT 带来了一套独特的设备管理方法，每个企业级解决方案都必须满足以下的原则，如图10.3所示。

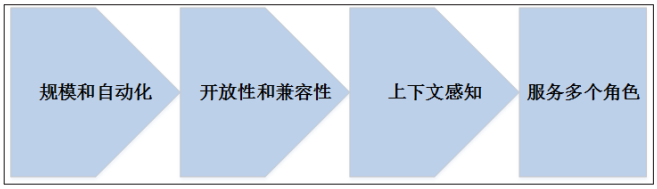


图 10.3　企业级解决方案必须满足的原则

(1) 规模和自动化：IoT 解决方案需要简单的工具，可以自动执行日常任务，并使相对较少的运营人员管理数百万台设备。平时，运营商希望能远程处理大量设备操作，并且仅在出现需要他们直接注意的问题时才提醒。

(2) 开放性和兼容性：设备生态系统非常多样化，管理工具必须定制以适应多种设备类、平台和协议。运营商必须能够支持多种类型的设备，从最受限制的嵌入式单进程芯片到功能强大和功能齐全的计算机。

(3) 上下文感知：IoT 环境是动态的、不断变化的，所以服务可靠性至关重要。设备管理操作必须考虑 SLA 的时间要求、网络和电源状态、使用条件和设备地理位置，以确保维护需要的时间不会影响关键业务操作或引起危险。

(4) 服务多个角色：支持独特的工作流程和流程的 IoT 操作角色至关重要。操作人员必须与内部 IT 部门的给定约束条件协调工作，还必须找到可持续的方法来向管理者和其他业务管理角色提供实时设备操作信息。

设备生命周期

这是一组设备管理阶段，对所有企业 IoT 项目都是通用的。在 Azure IoT 中，设备的生命周期有 5 个阶段，在每一个阶段中，需要满足以下几个针对设备操作员的要求以提供完整的解决方案。

(1) 计划：允许运营商创建设备元数据，使他们能够轻松、准确地查询和定向一组设备进行批量管理操作。可以使用设备双系统以标签和属性的形式存储此设备元数据。

(2) 激活：在 IoT Hub 里安全激活，并使操作员能够立即发现设备功能。使用 IoT Hub 身份注册表创建灵活的设备标识和凭据，并通过使用作业批量执行此操作。建立报表，通过设备双系统存储的设备属性报告其功能和条件。

(3) 配置：便于设备的批量配置更改和固件更新，同时保持设备正常工作和安全。通过使用所需的属性或使用直接方法和广播作业批量执行这些设备管理操作。

(4) 监视器：监视整体设备运行状况，正在进行的操作的状态，并警告操作员可能需要注意的问题。应用设备双系统允许设备报告实时操作条件和更新操作的状态。通过使用设备双查询来建立功能强大的仪表板报告展示最紧迫的问题。

(5) 淘汰：在发生故障、升级周期或服务生命周期结束后更换或停用设备。如果物理设备正在更换，则使用设备双系统维护设备信息；如果更换完成，则使用设备双系统进行归档。使用 IoT Hub 身份注册表可安全撤销设备标识和凭据。

设备管理模式

IoT Hub 支持以下设备管理模式。

(1) 重启设备：后端应用程序通过直接方法通知设备已进行重新启动。设备使用报告的属性更新设备的重新启动状态，如图10.4所示。

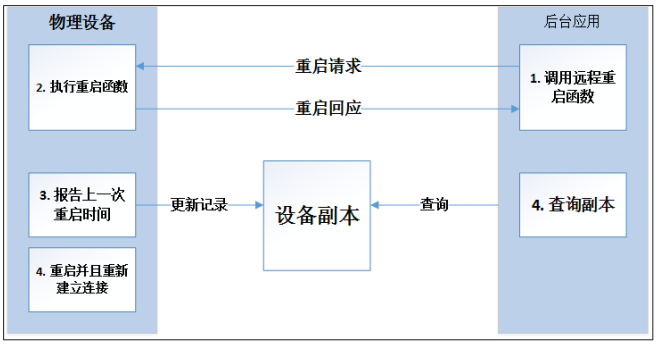


图 10.4　设备的重新启动状态更新

(2) 恢复出厂设置：后端应用程序通过直接方法通知设备已恢复出厂设置。设备使用报告的属性更新设备的出厂复位状态。恢复出厂设置的流程和重启设备的流程相同。

(3) 配置：后端应用程序使用所需要的属性来配置在设备上运行的软件。设备使用报告的属性更新设备的配置状态，如图10.5所示。

(4) 固件更新：后端应用程序通过直接方法通知设备已启动固件更新。设备启动多步过程以下载固件映像并应用固件映像，最后重新连接到 IoT Hub 服务。在整个多步骤过程中，设备使用所报告的属性来更新设备的进度和状态，如图10.6所示。

(5) 报告进度和状态：解决方案后端运行设备双查询，并跨一组设备，以报告在设备上运行的操作状态和进度。具体的流程图比较复杂，这里就不介绍了。

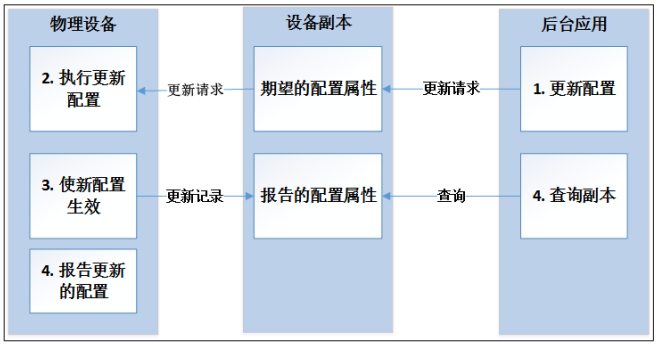


图 10.5　设备配置状态更新

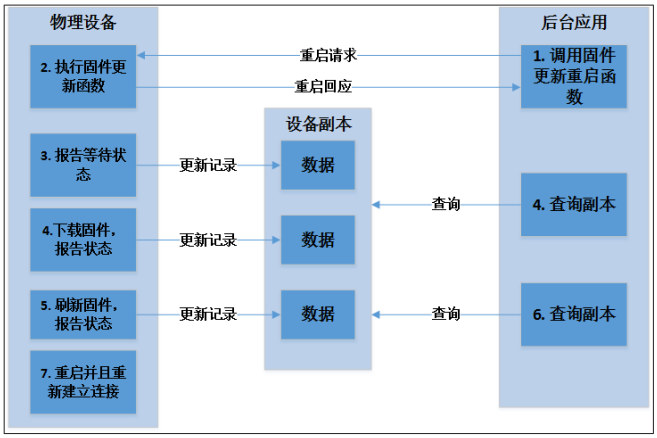


图 10.6　设备更新进度和状态

下一步

使用 IoT Hub 提供的功能、设备和代码库创建满足企业 IoT 运营商要求的 IoT 应用程序。

10.4　使用.NET 将模拟设备连接到 IoT 中心

简介

Azure IoT Hub 是一种完全托管的服务，可在数百万的 IoT 设备和解决方案后端之间实现可靠、安全的双向通信。IoT 项目面临的最大挑战之一是如何可靠、安全地将设备连接到解决方案后端。为了应对这一挑战，IoT Hub 能够：

(1) 提供可靠的设备到云和云到设备的超大规模消息。

(2) 使用每个设备的安全凭证和访问控制启用安全通信。

(3) 包含了最受欢迎的语言和平台的设备库。

下面在 Azure 上先创建 IoT Hub，并在这个 IoT Hub 上创建设备身份，然后创建一个模拟设备应用程序发送测量数据到解决方案后端，同时从后端接收命令。

为了完成这个任务，需要 Visual Studio 2015 和 Azure 账号。

建立 IoT Hub

(1) 登录 Azure 门户页面：https://portal.azure.com。

(2) 登录以后，依次点击图10.7中的标注框。

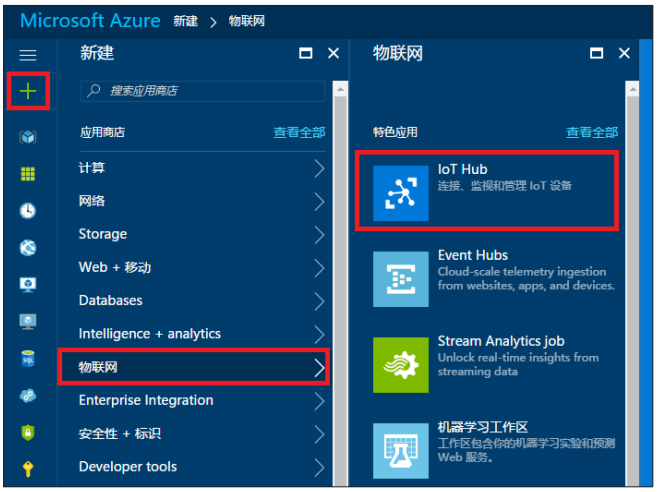


图 10.7 IoT 创建窗口

(3) 点击“IoT Hub”选项以后，选择以下配置：在名称框中输入 IoT Hub 的名称。如果名称有效而且没人注册过，则“名称”框中将显示一个绿色复选标记。选择定价和规模层，使用免费的 F1 层。在“资源”组中，创建资源组或选择现有注册好的资源组。在“位置”中，选择托管 IoT Hub 的位置，这里选择“美国西部”，如图10.8所示。

(4) 当完成配置选项以后，点击“创建”按钮。这个过程会花几分钟建立 IoT Hub。可以点击消息图标查看状态，如图10.9所示。

(5) 当 IoT Hub 建立好以后，在仪表板上点击新生成的 tile。记下主机名字，然后点击“共享访问策略”选项，如图10.10所示。



图 10.8　IoT 参数窗口

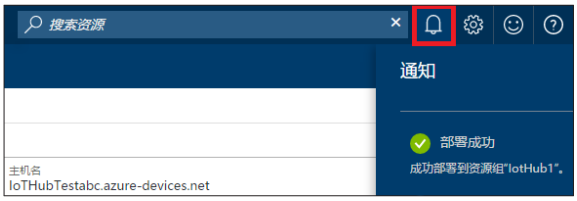


图 10.9　IoT 创建成功通知窗口

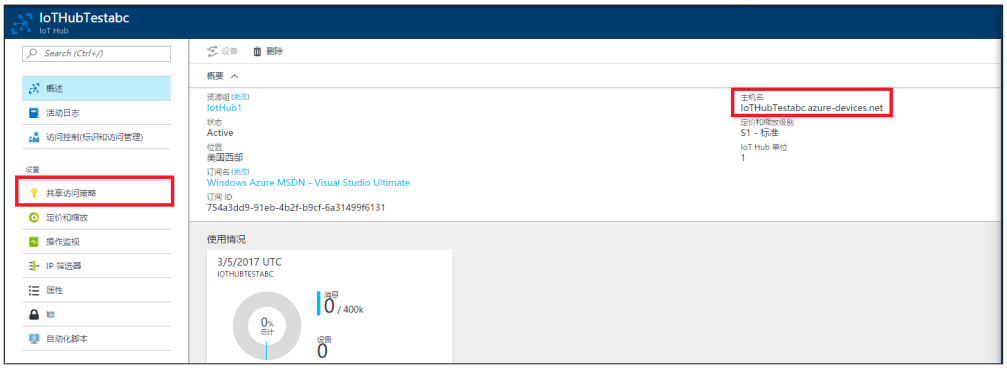


图 10.10　IoT 流量使用情况窗口

(6) 在共享访问策略里，点击“iotHubowner”选项，然后复制并记下 iotHubowner 中的 IoT Hub 连接字符串，如图10.11所示。



图 10.11　IoT 共享访问策略窗口

到此为止，IoTHub 创建完毕，接下来会用到主机名字和 IoT Hub 连接字符串。

创建设备身份

下面创建一个.NET 控制台应用程序，用于在 IoT Hub 的身份注册表中创建设备标识。只有设备在身份注册表中有记录时才能连接到 IoT Hub。当运行此控制台应用程序时，它会生成一个唯一的设备 ID 和密钥，当你的设备发送消息到 IoT Hub 时候需要标识它自己。

(1) 在 Visual studio 中，创建 Visual C# 控制台应用项目，命名为 CreateDeviceIdApp，如图10.12所示。

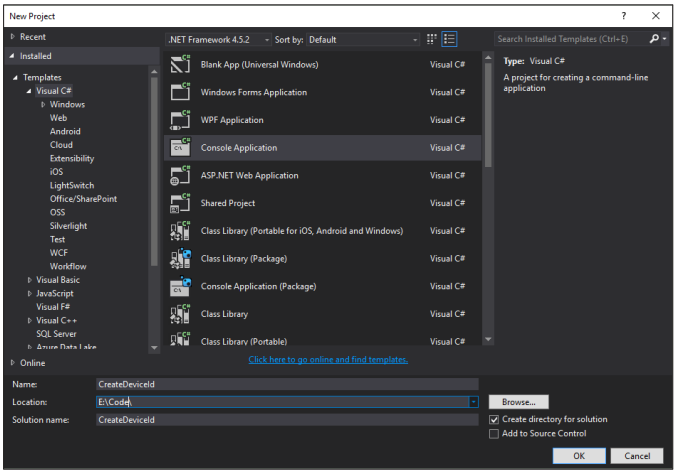


图 10.12　IoT 建立控制台程序窗口

(2) 从 NuGet 安装必需的库：右击 CreateDeviceId 项目，在弹出的快捷菜单中选择“Manage Nuget Package…”选项。搜索“microsoft.azure.devices”，选中返回的第一个选项，点击“Install”选项安装，如果提示需要 Accept Licences，点击“Accept”选项即可，如图10.13所示。

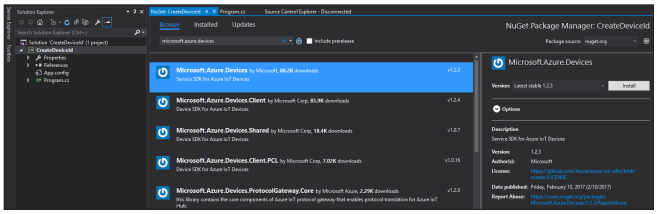


图 10.13　从 NuGet 安装必需的库

(3) 控制台程序入口在 CreateDeviceIdApp.cs 文件中，修改这个文件如下。

```
using System;
using System.Threading.Tasks;
using Microsoft.Azure.Devices;
using Microsoft.Azure.Devices.Common.Exceptions;
using System.Diagnostics;

namespace IoTHub.CreateDeviceId
{
    public class CreateDeviceIdApp
    {
        private readonly RegistryManager registryManager;
        private readonly string connectionString;
        private readonly string deviceId;

        public CreateDeviceIdApp(string devicdId, string connectionString)
        {
            this.deviceId = "TestDevice1";
            this.connectionString = connectionString;

            this.registryManager = RegistryManager.
            CreateFromConnectionString(this.connectionString);
        }

        static void Main(string[] args)
        {
            TraceListener consoleTraceListener = new System.Diagnostics.
            ConsoleTraceListener();
            Trace.Listeners.Add(consoleTraceListener);

```

```
28              CreateDeviceIdApp app = new CreateDeviceIdApp(
29                  "TestDevice1",
30                  "HostName=IoTHubTestabc.azure-devices.net;
                    SharedAccessKeyName=iotHubowner;SharedAccessKey=x9cO3s+
                    fp9Szpv9tjLxlngZrIcFViETQlgrMqXFuesM=");
31
32              app.AddDeviceAsync().Wait();
33              Console.ReadKey();
34          }
35
36          private async Task AddDeviceAsync()
37          {
38              Device device = null;
39              try
40              {
41                  device = await this.registryManager.AddDeviceAsync(new
                    Device(this.deviceId));
42                  Trace.WriteLine(string.Format("Generated device key: {0}",
                    device.Authentication.SymmetricKey.PrimaryKey));
43              }
44              catch (DeviceAlreadyExistsException)
45              {
46                  device = await this.registryManager.GetDeviceAsync(this.
                    deviceId);
47                  Trace.WriteLine(string.Format("Fetch existing device key:
                    {0}", device.Authentication.SymmetricKey.PrimaryKey));
48              }
49              catch (Exception e)
50              {
51                  Trace.WriteLine(string.Format("Unexpected exception {0}", e)
                    );
52              }
53          }
54      }
55  }
```

(4) 按 Ctrl+F5 组合键执行程序，输出如图10.14所示，这个设备的密匙是：

```
LRTc+WYkVxLQNsPeVgOKZZjbI1RVqQhLOcw47IV9J8M=
```

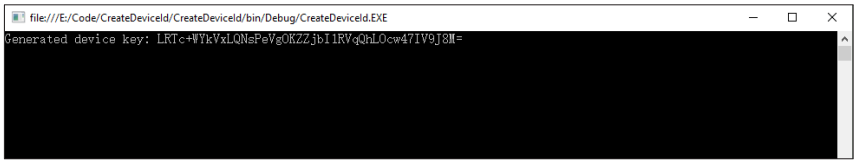


图 10.14　IoT 设备密匙

建立虚拟的设备

下面建立另一个控制台程序 VirtualDeviceApp，模拟一个设备发送消息到云端，创建过程类似前一个 Project，这里需要安装不同的 Nuget 库，如图10.15所示。

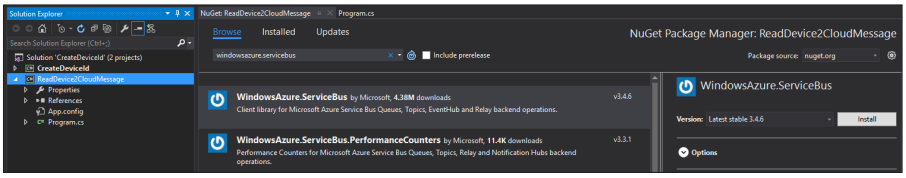


图 10.15　创建 VirtualDeviceApp

然后修改 VirtualDeviceApp.cs 如下。

```
using System;
using System.Text;

using Microsoft.Azure.Devices.Client;
using Newtonsoft.Json;
using System.Diagnostics;

namespace IoTHub.VirtualDevice
{
    public class VirtualDeviceApp
    {
        private static Random Rand = new Random();
        private readonly DeviceClient deviceClient;
        private string deviceId;
        private string iotHubUri;
        private string deviceKey;

        public VirtualDeviceApp(string deviceId, string iotHubUri, string
        deviceKey)
```

```
19          {
20              this.deviceId = deviceId;
21              this.iotHubUri = iotHubUri;
22              this.deviceKey = deviceKey;
23              this.deviceClient = DeviceClient.Create(
24                  this.iotHubUri,
25                  new DeviceAuthenticationWithRegistrySymmetricKey(this.
                    deviceId, this.deviceKey), TransportType.Mqtt);
26          }
27
28          static void Main(string[] args)
29          {
30              TraceListener consoleTraceListener = new System.Diagnostics.
                ConsoleTraceListener();
31              Trace.Listeners.Add(consoleTraceListener);
32
33              VirtualDeviceApp deviceApp = new VirtualDeviceApp(
34                  "TestDevice1",
35                  "IoTHubTestabc.azure-devices.net",
36                  "LRTc+WYkVxLQNsPeVgOKZZjbI1RVqQhLOcw47IV9J8M=");
37
38              Console.WriteLine("Virtual device is created.\n");
39              deviceApp.SendDevice2CloudMessagesAsync();
40              Console.ReadKey();
41
42          }
43
44          private async void SendDevice2CloudMessagesAsync()
45          {
46              double temperatureInCelSius = 30;
47              while (true)
48              {
49                  double temperature = temperatureInCelSius + Rand.NextDouble
                    () * 5;
50                  var dataSample = new
51                  {
52                      deviceId = this.deviceId,
```

```
                    temperature = temperature,
                    guid = Guid.NewGuid().ToString()
                };

                string messageString = JsonConvert.SerializeObject(
                dataSample);
                Message message = new Message(Encoding.ASCII.GetBytes(
                messageString));

                await this.deviceClient.SendEventAsync(message);

                Trace.WriteLine(string.Format("{0}, Sending message: {1}",
                DateTime.Now, messageString));
            }
        }
    }
}
```

接收设备到云端消息

下面创建另一个简单的控制台程序 ReadDevice2CloudMessage，从 IoT Hub 中读取设备到云端消息。创建过程类似于前一个 Project，这里需要安装不同的 NuGet 库，如图10.16所示。

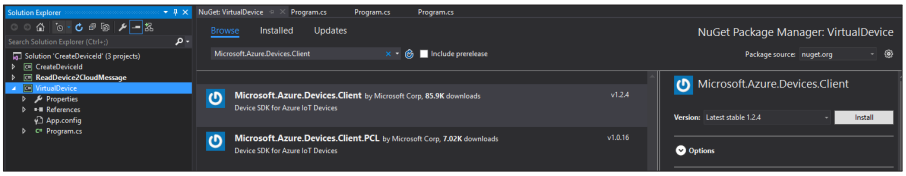


图 10.16　创建 ReadDevice2CloudMessage

然后修改 ReadDevice2CloudMessageApp.cs：

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading.Tasks;
using Microsoft.ServiceBus.Messaging;
using System.Threading;
```

```
using System.Diagnostics;

namespace IotHub.ReadDevice2CloudMessage
{
    public class ReadDevice2CloudMessageApp
    {
        private readonly string connectionString;
        private readonly string iotHubD2cEndpoint;
        private readonly EventHubClient eventHubClient;

        public ReadDevice2CloudMessageApp(string connectionString, string
        iotHubD2cEndpoint)
        {
            this.connectionString = connectionString;
            this.iotHubD2cEndpoint = iotHubD2cEndpoint;
            this.eventHubClient = EventHubClient.CreateFromConnectionString(
            this.connectionString, this.iotHubD2cEndpoint);
        }

        static void Main(string[] args)
        {
            TraceListener consoleTraceListener = new System.Diagnostics.
            ConsoleTraceListener();
            Trace.Listeners.Add(consoleTraceListener);

            ReadDevice2CloudMessageApp app = new ReadDevice2CloudMessageApp(
                "HostName=IoTHubTestabc.azure-devices.net;
                SharedAccessKeyName=iotHubowner;SharedAccessKey=
                YtRvhBpllem5j6twJ2PIUmXYCT2zeiSoqOXYitqR2kY=",
                "messages/events");

            string[] d2cPartitions = app.eventHubClient.
            GetRuntimeInformation().PartitionIds;

            CancellationTokenSource cts = new CancellationTokenSource();

            List<Task> tasks = new List<Task>();
```

```
38              foreach (string partition in d2cPartitions)
39              {
40                  tasks.Add(app.ReceiveDevice2CloudMessagesAsync(partition,
                    cts.Token));
41              }
42              Task.WaitAll(tasks.ToArray());
43          }
44
45          private async Task ReceiveDevice2CloudMessagesAsync(string partition
            , CancellationToken ct)
46          {
47              EventHubReceiver eventHubReceiver = this.eventHubClient.
                GetDefaultConsumerGroup().CreateReceiver(partition, DateTime.
                UtcNow);
48              while (true)
49              {
50                  if (ct.IsCancellationRequested) break;
51                  EventData eventData = await eventHubReceiver.ReceiveAsync();
52                  if (eventData == null) continue;
53
54                  string data = Encoding.UTF8.GetString(eventData.GetBytes());
55                  Trace.WriteLine(string.Format("Message received. Partition:
                    {0} Data: '{1}'", partition, data));
56              }
57          }
58      }
59  }
```

运行程序

现在我们有了发送消息程序和读取消息程序，只需要将这两个 APP 同时启动，右击 IotHubTest.sln，在弹出的快捷菜单中把 ReadDevice2CloudMessage 和 VirtualDevice 这两个 project 标记为 Startup，如图10.17所示。

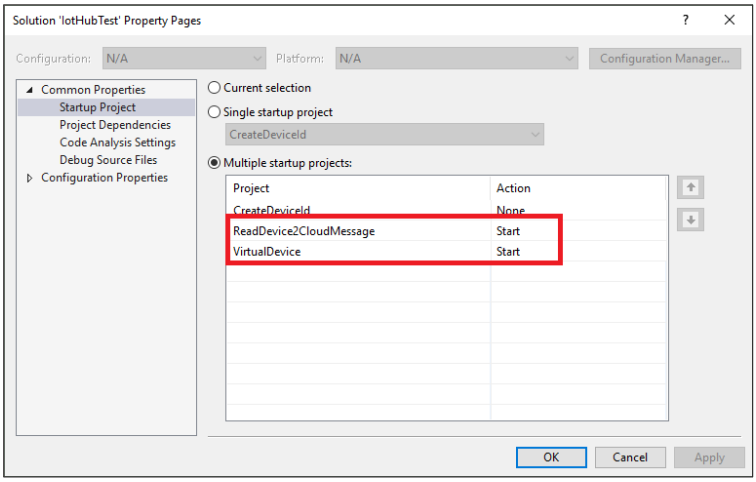


图 10.17　创建 ReadDevice2CloudMessage

在按 F5 键的同时启动两个程序，会看到有两个控制台窗口生成：VirtualDeviceApp 窗口显示了设备向 IoT Hub 发送的命令，ReadDevice2cloudMessage 窗口（见图10.18）显示了从 IoT Hub 中读到的命令。

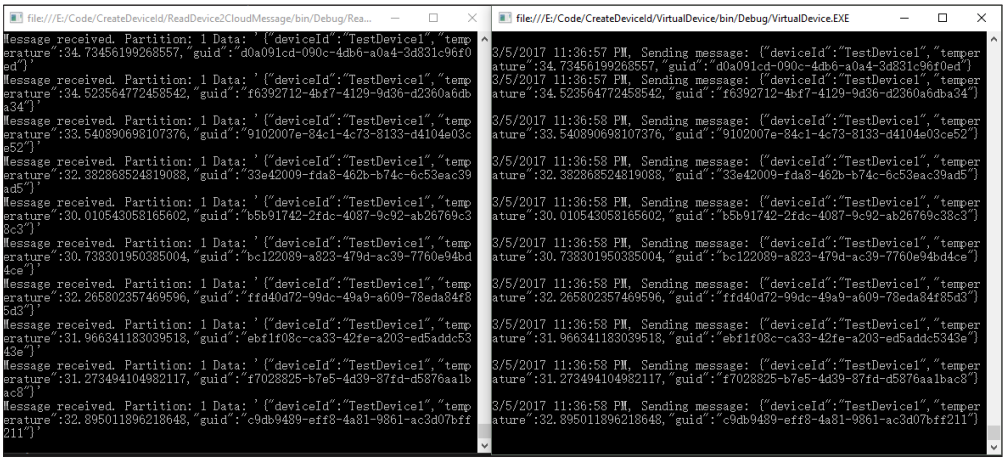


图 10.18　向 IoT Hub 发送命令和接受命令

而且，在 Azure 门户网站上可以看到这个 IoT Hub 的流量使用情况（见图10.19）。

使用 Service Bus 缓存命令

在以上程序中，当 IoT Hub 接到命令以后立刻就被读到，没有缓存或者队列，所以其不适合大数量的设备的部署，因为那样会要求 IoT Hub 有高吞吐。为了应对这样的场景，Azure 引入了 Service Bus 的概念，下面介绍 Service Bus 的例子程序。

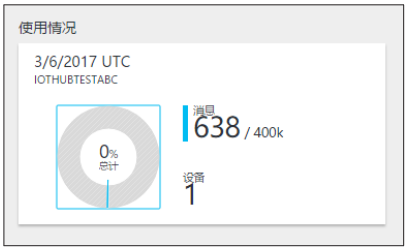


图 10.19　IoT Hub 流量使用情况

首先在 Azure Portal 里安装 Service Bus。登录 Azure Portal，点击加号按钮，选择“Enterprise Integration”，然后在右侧的下拉菜单里选择“Service Bus”选项，如图10.20所示。

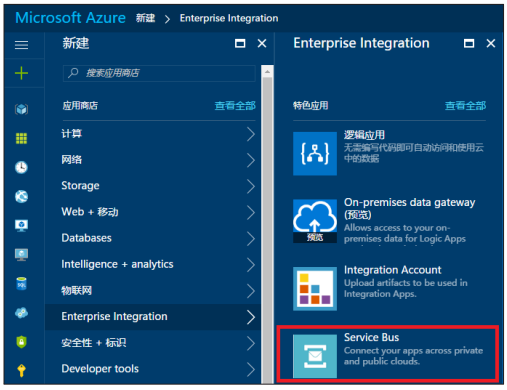


图 10.20　Service bus 选项

在图10.21所示的界面中填入参数，点击“创建”按钮，同时选择固定在仪表板上，这样可以快速访问。

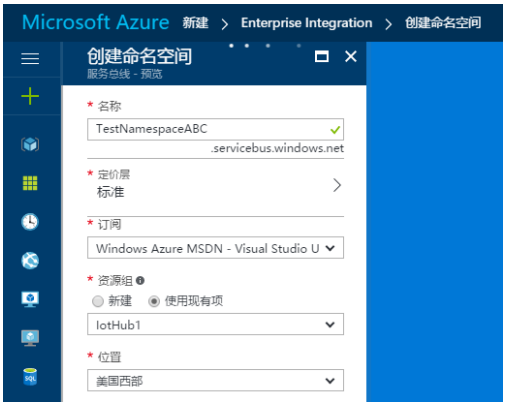


图 10.21　创建命名空间

打开新建的namespace，点击共享访问策略，然后点击“RootManageSharedAccessKey”，并记下以下内容。

- 主密钥：

  ```
  ISt+pe4OjDD+qbUaNx59wsNPCVT46VMWIiYbZt4UqMg=
  ```

- 连接字符串：

  ```
  Endpoint=sb://testnamespaceabc.servicebus.windows.net/;Shared
  AccessKeyName=RootManageSharedAccessKey;SharedAccessKe
  y=ISt+pe4OjDD+qbUaNx59wsNPCVT46VMWIiYbZt4UqMg=
  ```

然后可以建立一个队列。跳回到共享访问策略页面，点击“队列”选项，如图10.22所示，填入以下参数，点击“创建”选项。

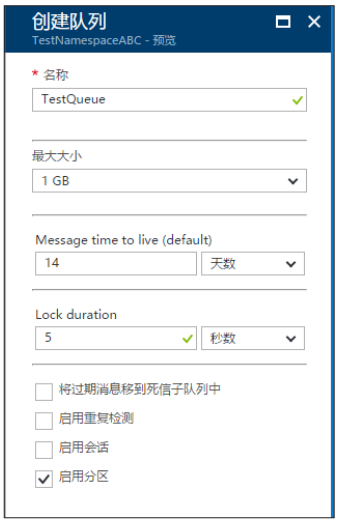


图 10.22 创建队列

接着，我们就可以写代码发送消息了。按照之前的步骤，创建一个控制台程序，首先安装 WindowsAzure.ServiceBus nuget 包，然后编辑 ServiceBusQueueTest.cs。

```
1  namespace SendServiceBusQueue {
2
3      using System;
4      using Microsoft.ServiceBus.Messaging;
5
6      public class SendServiceBusQueue
7      {
8          static void Main(string[] args)
```

```
        {
            string connectionString = "End-point=sb://testnamespaceabc.
            servicebus.windows.net/;SharedAccessKeyName=
            RootManageSharedAccessKey;SharedAccessKey=ISt+pe4OjDD+
            qbUaNx59wsNPCVT46VMWIiYbZt4UqMg=";
            string queueName = "TestQueue";

            QueueClient client = QueueCli-ent.CreateFromConnectionString(
            connectionString, queueName);
            BrokeredMessage message = new BrokeredMessage("This is a test
            mes-sage!");
            client.Send(message);

            Console.ReadKey();
        }
    }
}
```

运行程序 3 次，然后刷新 Azure Portal 中的 TestQueue 队列的信息，活动消息计数已经更新为 3，如图10.23所示。

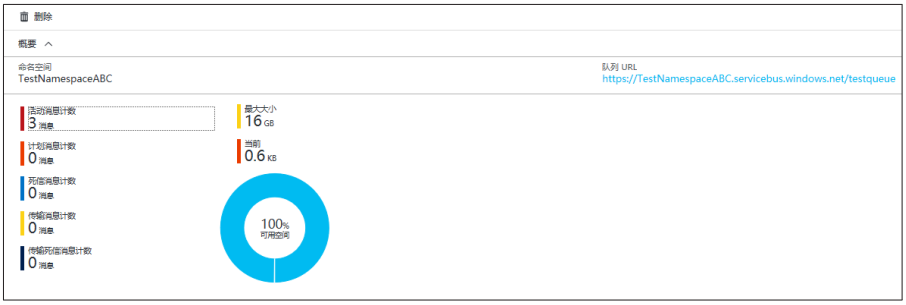


图 10.23　活动消息计数

接下来，再写一个程序从队列里读消息：

```
namespace ReadFromServiceBusQueue
{
    using System;
    using Microsoft.ServiceBus.Messaging;

    public class Program
    {
```

```
        public static void Main(string[] args)
        {
            string connectionString = "Endpoint=sb://testnamespaceabc.
            servicebus.windows.net/;SharedAccessKeyName=
            RootManageSharedAccessKey;SharedAccessKey=ISt+pe4OjDD+
            qbUaNx59wsNPCVT46VMWIiYbZt4UqMg=";
            string queueName = "TestQueue";

            QueueClient client = QueueClient.CreateFromConnectionString(
            connectionString, queueName);

            client.OnMessage(message =>
            {
                Console.WriteLine($"Message id: {message.MessageId}");
                Console.WriteLine($"Message body: {message.GetBody<String>()
                }");

            });

            Console.ReadKey();
        }
    }
}
```

运行程序，控制台输出 3 条消息，如图10.24所示。



图 10.24 控制台输出

然后刷新 TestQueue 队列活动消息计数已经变成 0，如图10.25所示。

连接 IoT Hub 和队列

上面分别介绍了 IoT Hub 和队列的各自程序，现在再写一个程序把满足特定条件的消息发送到这个队列里。

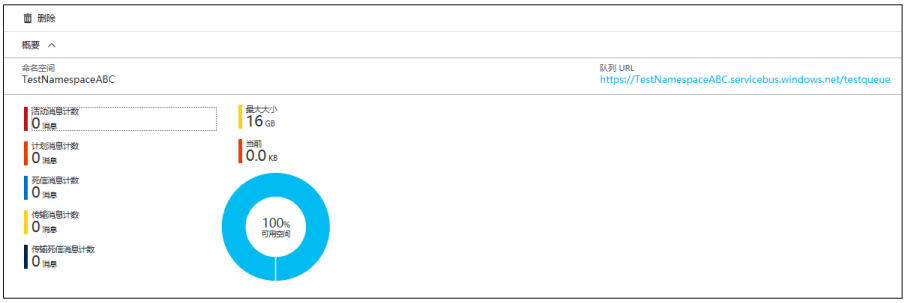


图 10.25　队列消息计数

(1) 在之前建立的 VirtualDeviceApp.cs 生成的数据样本中包含随机的温度，现在在这里加一个属性 Bucket 表明温度高低，如果温度高于 33 摄氏度，则 bucket=high，不然 bucket=low。具体程序如下（只改动了 SendDevice2CloudMessagesAsync 函数部分）：

```
private async void SendDevice2CloudMessagesAsync()
        {
            double temperatureInCelSius = 30;

            while (true)
            {
                double temperature = temperatureInCelSius + Rand.NextDouble
                () * 5;

                var dataSample = new
                {
                    deviceId = this.deviceId,
                    temperature = temperature,
                    guid = Guid.NewGuid().ToString()
                };

                string messageString = JsonConvert.SerializeObject(
                dataSample);
                string bucket;
                if (temperature > 33f)
                {
                    messageString = "This is a high temperature";
                    bucket = "high";
```

```
                }
                else
                {
                    bucket = "normal";
                }

                Message message = new Message(Encoding.ASCII.GetBytes(
                messageString));
                message.Properties.Add("bucket", bucket);

                await this.deviceClient.SendEventAsync(message);

                Trace.WriteLine($"{DateTime.Now}, Sending message: {
                messageString}");
            }
        }
```

(2) 在 Azure Portal 里，打开之前建立的 IOTHubTestabc，点击“终结点”选项，然后点击“添加”选项，如图10.26所示，建立终结点。

(3) 然后点击左侧的“路由”选项，如图10.27所示，添加路由。



图 10.26 添加终结点

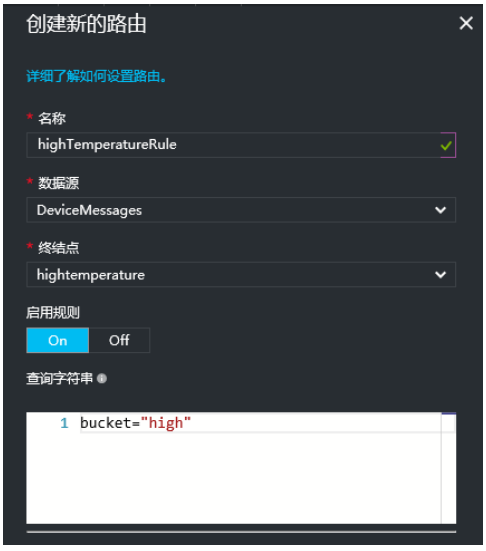


图 10.27 创建新路由

(4) 建立一个新程序从队列里读取消息，程序如下：

```
namespace ConsoleApplication1
{
    using System.IO;
    using System;
    using System.Text;

    using Microsoft.ServiceBus.Messaging;

    class Program
    {
        static void Main(string[] args)
        {
            string connectionString = "Endpoint=sb://testnamespaceabc.
            servicebus.windows.net/;SharedAccessKeyName=
            RootManageSharedAccessKey;SharedAccessKey=ISt+pe4OjDD+
            qbUaNx59wsNPCVT46VMWIiYbZt4UqMg=";
            string queueName = "TestQueue";

            QueueClient client = QueueClient.CreateFromConnectionString(
            connectionString, queueName);

            client.OnMessage(message =>
            {
                Stream stream = message.GetBody<Stream>();
                StreamReader reader = new StreamReader(stream, Encoding.
                ASCII);
                string s = reader.ReadToEnd();
                Console.WriteLine($"Message id: {message.MessageId}");
                Console.WriteLine($"Message body: {s}");

            });

            Console.ReadKey();
        }
    }
}
```

然后同时运行 VirtualDevice、ReadDevice2CloudMessage 和 ReadHighTemperature-Queue。第三个窗口显示了队列只收到了温度 > 33 的消息，如图10.28所示。

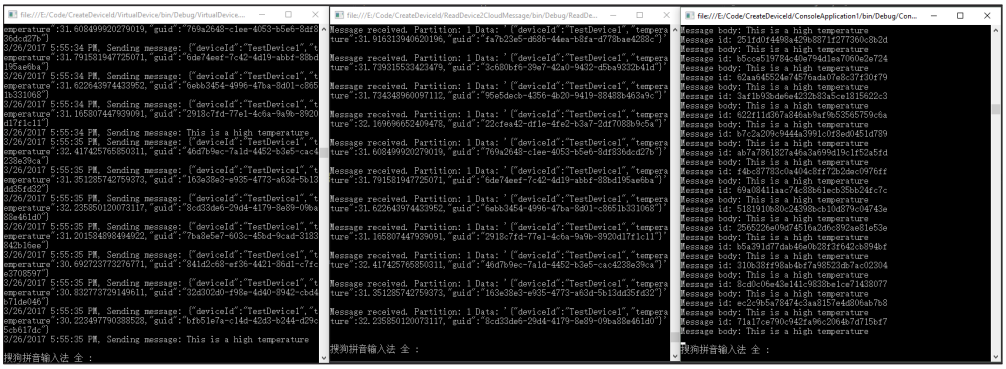


图 10.28　三个程序同时运行输出窗口

10.5　机器学习应用实例

在10.4节中介绍了如何用 IOT 数据，本节介绍用机器学习的方法从数据中获取价值。我们用 Azure 机器学习工作室来（Azure Machine Learning Studio）完成这个任务。首先介绍下这个工作室。

微软的 Azure 机器学习工作室是一种协作式的拖放工具，可用于构建、测试和部署数据上的预测分析解决方案。工作室可以将训练好的模型作为 Web 服务发布。在工作室里，写程序不是必要的，大部分情况只需要连接数据集和模块构建模型。

Azure 机器学习的链接地址是 https://studio.azureml.net/，登录以后，点击“PROJECTS”选项创建一个项目，取名为“IOT Test Project”，然后点击“EXPERIMENTS”创建一个实验，取名为“Temperature Anomaly Detection”（温度异常检测），然后把实验加入项目，结果如图10.29所示。

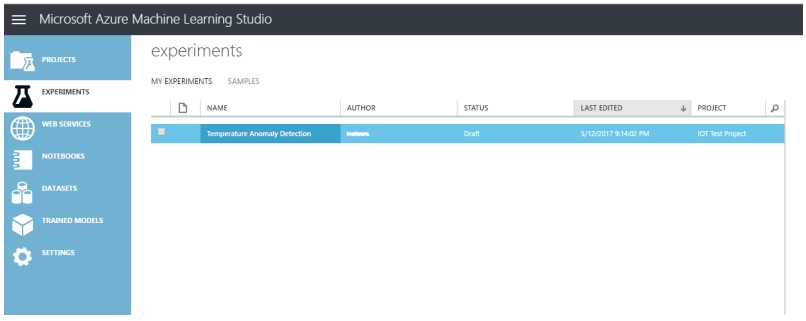


图 10.29　Azure 机器学习工作室界面

接下来我们完成这个实验。这个例子主要解释如何使用 Azure Machine Learning，所以不用 IoT 传感器输入的温度，而是假设 IoT 的设备温度已经存到了名 Time_Temperature.csv 的 excel 文件里，一共有 1440 条数据，每条数据有两维：时间和温度，如下所示。

```
      Time                          Temperature
0     2000-01-01 00:00:00+00:00     21.5
1     2000-01-02 00:00:00+00:00     27.4
2     2000-01-03 00:00:00+00:00     29.3
3     2000-01-04 00:00:00+00:00     35.0
4     2000-01-05 00:00:00+00:00     34.1
5     2000-01-06 00:00:00+00:00     30.3
6     2000-01-07 00:00:00+00:00     25.0
7     2000-01-08 00:00:00+00:00     24.6
8     2000-01-09 00:00:00+00:00     29.7
9     2000-01-10 00:00:00+00:00     25.3
10    2000-01-11 00:00:00+00:00     23.3
11    2000-01-12 00:00:00+00:00     24.7
...                        ...      ...
1431 2003-12-02 00:00:00+00:00      25.1
1432 2003-12-03 00:00:00+00:00      28.2
1433 2003-12-04 00:00:00+00:00      22.9
1434 2003-12-05 00:00:00+00:00      22.0
1435 2003-12-06 00:00:00+00:00      31.5
1436 2003-12-07 00:00:00+00:00      33.8
1437 2003-12-08 00:00:00+00:00      31.1
1438 2003-12-09 00:00:00+00:00      25.4
1439 2003-12-10 00:00:00+00:00      33.6
```

然后打开实验，在左侧搜索框里搜出三个模块并拖到中央，依次连接。图10.30展示了这样操作的一个截图。

中间的“Execute Python Script”是由开发者编写逻辑，在这个例子里，我们用正态分布模拟温度的分布，把在 3 倍标准差以外的温度定义成异常温度。点击这个模块，右侧出现代码编辑框，这个 Python 程序如下：

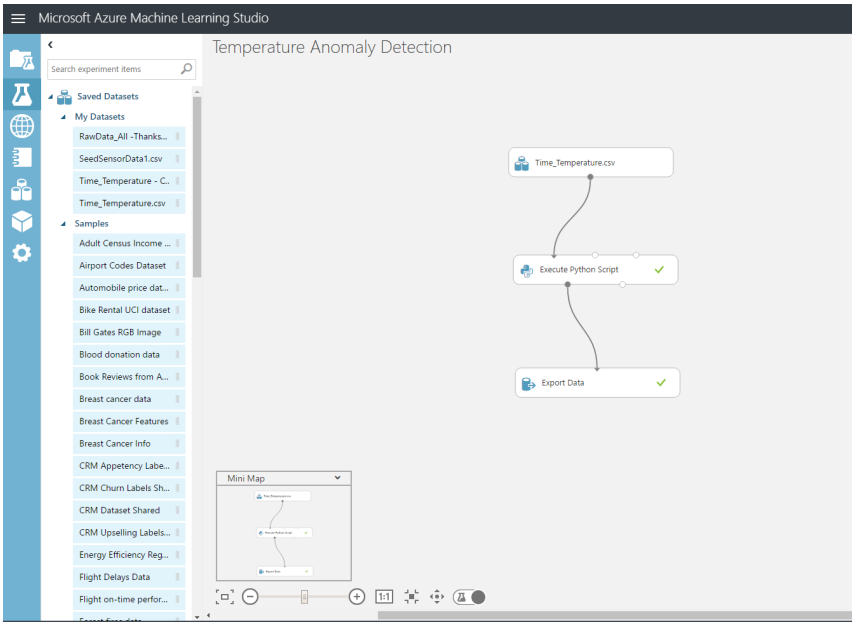


图 10.30 构造 AzureML 实验

```
# The script MUST contain a function named azureml_main
# which is the entry point for this module.

# imports up here can be used to
import pandas as pd

# The entry point function can contain up to two input arguments:
#   Param<dataframe1>: a pandas.DataFrame
#   Param<dataframe2>: a pandas.DataFrame
def azureml_main(dataframe1 = None, dataframe2 = None):

    # Execution logic goes here
    print('Input pandas.DataFrame #1:\r\n\r\n{0}'.format(dataframe1))

    # 正态分布数据样本大小
    windowsize =120

    # 3倍sigma
    multiplier = 3
```

```
20
21      # 根据给定的样本，计算方差
22      rollingstd = pd.rolling_std(dataframe1.Temperature, window=windowsize)
23
24      # 根据给定的样本，计算期望值
25      rollingmean = pd.rolling_mean(dataframe1.Temperature, window=windowsize)
26
27      # 计算非异常温度的区间
28      ucl = rollingmean + multiplier*rollingstd
29      lcl = rollingmean - multiplier*rollingstd
30
31      # 标注温度的异常与否（true 表示正常， false 表示异常）
32      dataframe1["Alert"] = (dataframe1.Temperature > ucl) | (dataframe1.
        Temperature < lcl )
33
34      # If a zip file is connected to the third input port is connected,
35      # it is unzipped under ".\Script Bundle". This directory is added
36      # to sys.path. Therefore, if your zip file contains a Python file
37      # mymodule.py you can import it using:
38      # import mymodule
39
40      # 输出样本，比输入样本多一列
41      return dataframe1,
```

然后，需要配置输出 csv 文件的目的地。点击“Export Data”选项，右侧会出现配置框，我们选择保存到 Azure 的 Blob，如图10.31所示。存储账户需要提前设置，这里就不赘述了。

然后点击“Run”按钮运行，按钮会变灰表示正在运行，一旦变回可用，说明运行完成。从 Azure blob 里下载输出的 csv 文件，从下面的结果中可以看到比输入多的一列就是我们想要的结果。

```
1  Time            Temperature     Alert
2  1/1/2000 0:00   21.5            FALSE
3  1/2/2000 0:00   27.4            FALSE
4  1/3/2000 0:00   29.3            FALSE
5  1/4/2000 0:00   35              FALSE
6  1/5/2000 0:00   34.1            FALSE
```

```
1/6/2000 0:00   30.3            FALSE
1/7/2000 0:00   25              FALSE
1/8/2000 0:00   24.6            FALSE
1/9/2000 0:00   29.7            FALSE
1/10/2000 0:00  25.3            FALSE
... ... ...
11/30/2003 0:00 31.1            FALSE
12/1/2003 0:00  26.2            FALSE
12/2/2003 0:00  25.1            FALSE
12/3/2003 0:00  28.2            FALSE
12/4/2003 0:00  22.9            FALSE
12/5/2003 0:00  22              FALSE
12/6/2003 0:00  31.5            FALSE
12/7/2003 0:00  33.8            FALSE
12/8/2003 0:00  31.1            FALSE
12/9/2003 0:00  25.4            FALSE
12/10/2003 0:00 33.6            FALSE
```

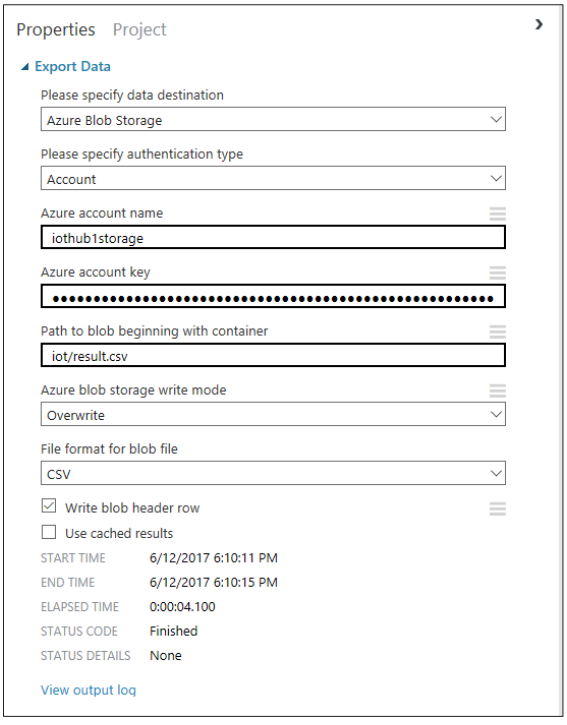


图 10.31 配置 csv 文件输出目的地

用 Excel 内置的画图工具将数据可视化，结果如图10.32所示，可以看到，在期望值±3 倍标准差以外的点都被识别成异常温度。

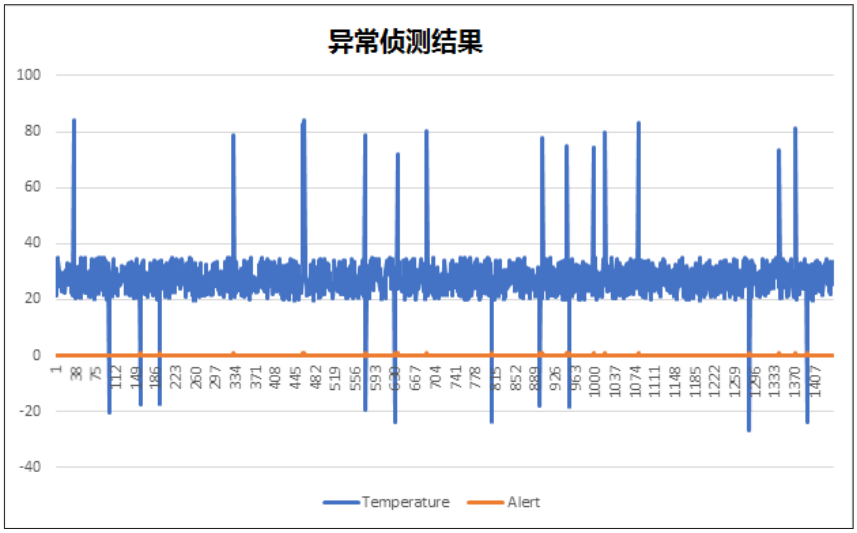


图 10.32　异常温度侦测结果可视化

下面讲述如何从 IoT Hub 中采集到数据、存储，然后传给 Azure 机器学习工作室。有以下几个准备工作要做。

(1) 创建 Azure 的存储账号。

(2) 为 IoT Hub 连接准备读取消息。

(3) 新建 Azure 的函数应用。

首先，创建 Azure 的存储账号。在 Azure 的首页中点击“新建”—“Storage”—选项创建存储账户，如图10.33所示。

其次，需要为 IoT Hub 连接准备读取消息。IoT Hub 内建的“事件中心兼容终结点”程序可以读取 IoT Hub 的消息。同时，这个程序用“使用者组”从 IoT Hub 读取消息。

先拿到 IoT Hub 终结点的连接字符串：打开 IoT Hub，点击“终结点”选项，然后点击“Events”，会出现如图10.34所示的界面，在最右边的面板里找到“事件中心-兼容名称”和“事件中心-兼容终结点”。

其中：

- 事件中心—兼容名称：iotHubtestabc。
- 事件中心—兼容终结点：Endpoint=sb://iotHub-ns-iotHubtest-123079-9af8d449d6.servicebus.windows.net/;SharedAccessKeyName=iotHubowner;SharedAccessKey=x9cO3s+fp9Szpv9tjLxlngZrIcFViETQlgrMqXFuesM=。

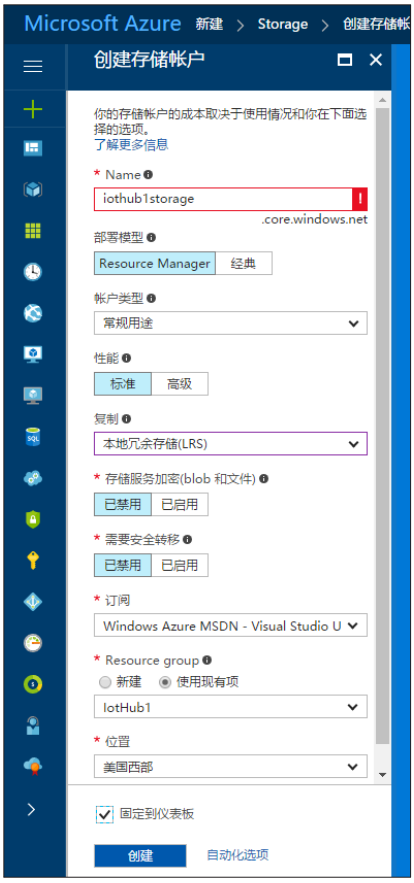


图 10.33　创建存储账号

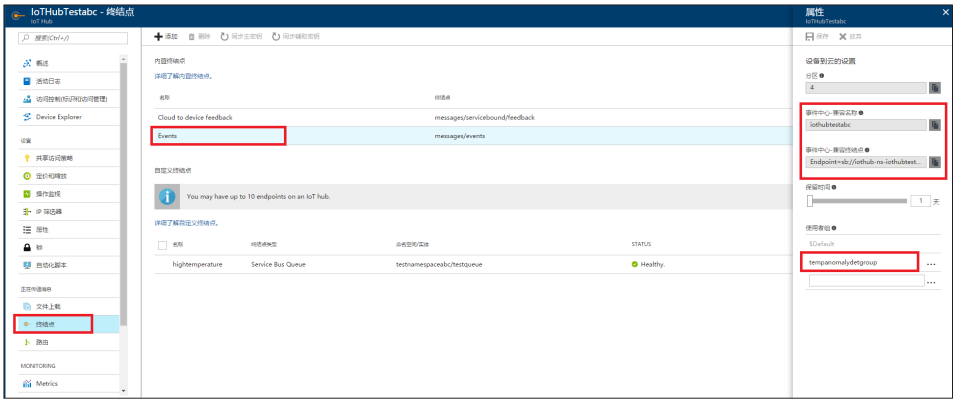


图 10.34　创建存储账号

在 IoT Hub 面板里，点击“共享访问策略”选项，在打开的页面中点击“iotHubowner”选项，在右侧面板里找到主密钥并记下来。这里的主密钥为：x9cO3s+fp9Szpv9tjLxlngZrIcFViETQlgrMqXFuesM=，如图10.35所示。要验证这个主密钥和事件中心—兼容终结点页面里的 SharedAccessKey 是否一致。如果不一致，则用前者替换后者里的 SharedAccessKey 值。

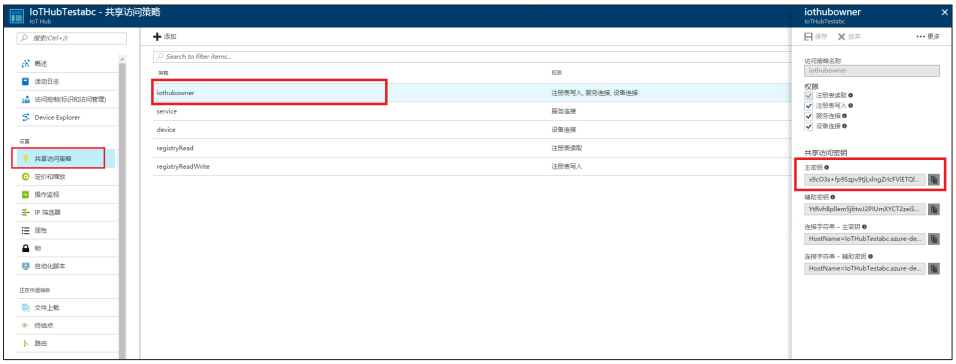


图 10.35　共享访问策略

为 IoT Hub 创建使用者群组。还是在最右侧面板里输入“tempanomalydet”，点击“保存”按钮即可，如图10.36所示。

接着需要创建 Azure 函数应用。在 Azure 首页，点击“新建”—“计算”—“函数应用选项，将应用取名为“iotHubtempconvert”，选择现有的资源组 IotHub1，选择之前建立的存储账号 iotHub1storage，点击“创建”按钮，展示效果如图10.37所示。

一旦新的函数应用创建完毕，就会出现如图10.38所示界面，点击“函式”选项，在页面右侧选择语言为“Javascript”，案例为“资料处理”，并选择 EventHubTrigger-JavaScript 模板。命名为“EventHubTriggerJS1”，将“事件中枢名称”（Event Hub）输入“iotHubtestabc”，为“事件中枢连线”（Event Hub connection）选择建立好 IoT Hub 终结点（只有一个选择），见图10.39。

接着，点击“建立”按钮。创建成功以后，点击“整合”选项，在打开的页面中点击“新的输出”选项，选择“Azure 资料表存储体”（Azure Table Storage），点击“选取”按钮。然后输入以下信息（见图10.40），这些信息会在代码中用到：

- 资料表参数名称（Table parameter name）：outputTable。
- 资料表名称（Table name）：temperatureData。
- 存储体账户连线（Storage account connection）：iotHub1storage_STORAGE。



图 10.36　使用者群组的创建

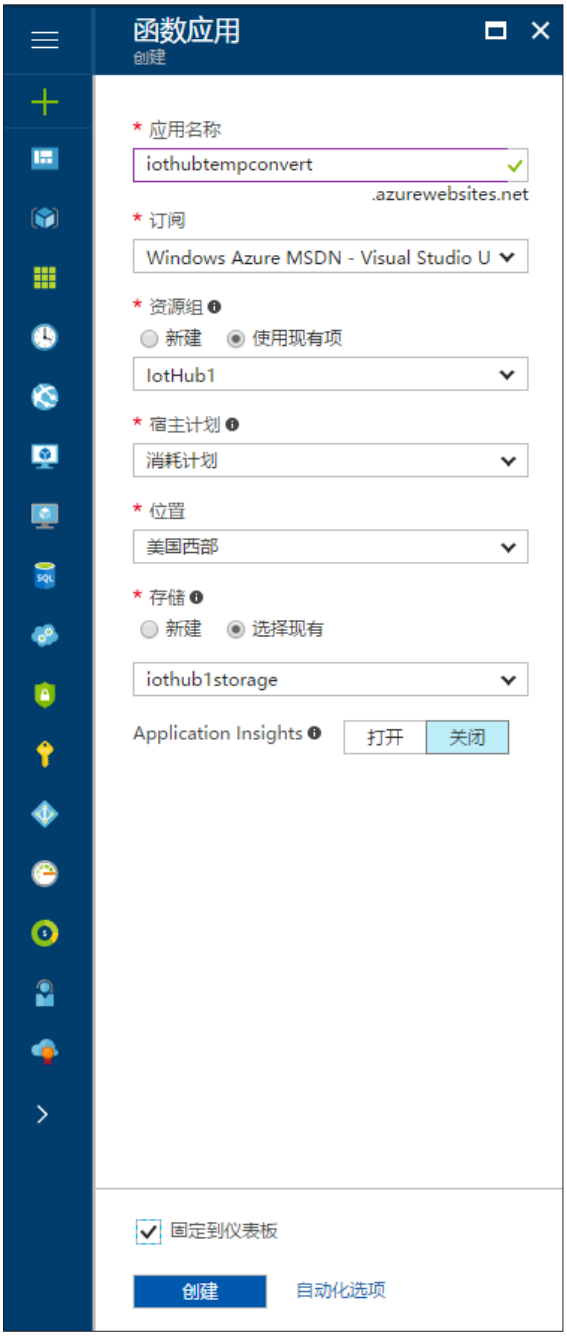


图 10.37　Azure 函数应用

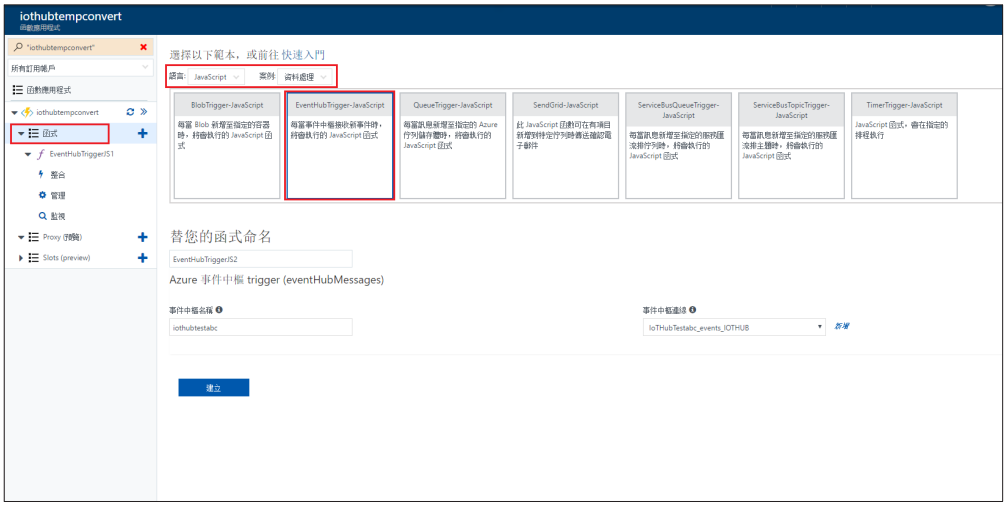


图 10.38　EventHubTrigger-JavaScript 模板

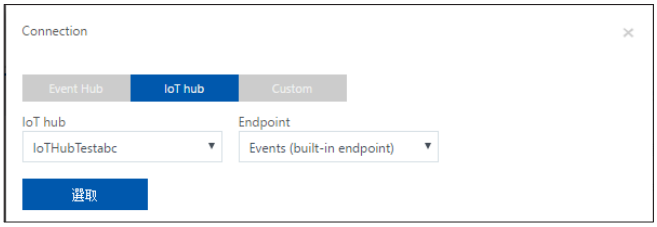


图 10.39　使用者群组的创建

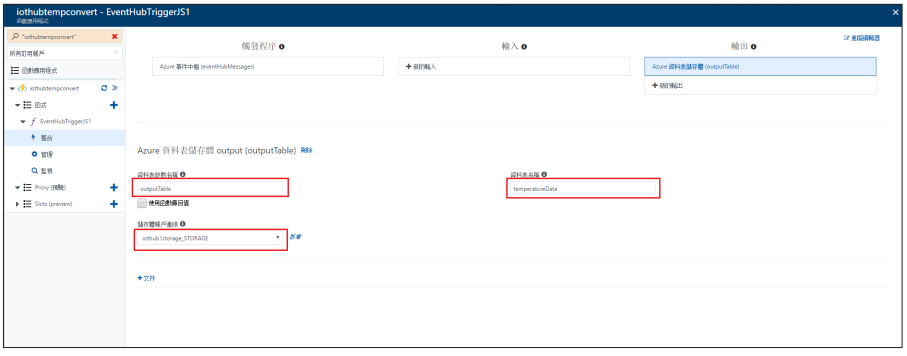


图 10.40　Azure 资料表存储体

然后点击“储存”按钮。会出现 Azure 事件中枢 Trigger（eventHubMessages）界面，输入图10.41所示的信息，注意“事件中枢取用者群组”的值是之前创建的“使用者群组”。

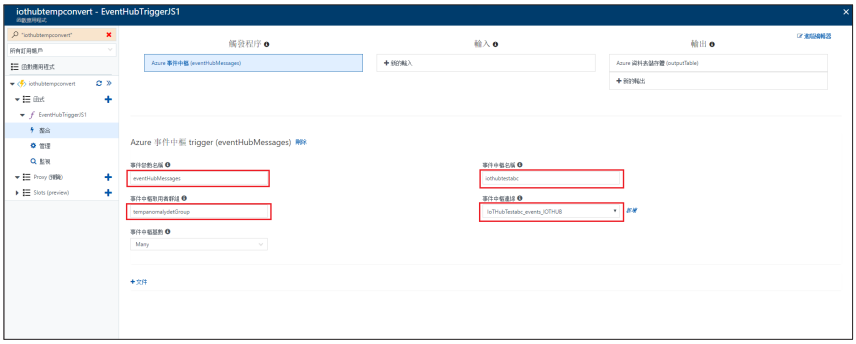


图 10.41　事件中枢

到此，可以开始编辑 JavaScript 代码了。点击左侧面板的“EventHubTriggerJS1”选项，出现代码编辑窗口，输入以下代码：

```
'use strict';

// This function is triggered each time a message is revieved in the IoTHub.
// The message payload is persisted in an Azure Storage Table

module.exports = function (context, iotHubMessage) {
 context.log('Message received: ' + JSON.stringify(iotHubMessage));
 var date = Date.now();
 var partitionKey = Math.floor(date / (24 * 60 * 60 * 1000)) + '';
 var rowKey = date + '';

 context.bindings.outputTable = {
    "partitionKey": partitionKey,
    "rowKey": rowKey,
    "Temperature": iotHubMessage[0].temperature
 }

 context.done();
};
```

保存以后，在右侧的测试窗口中输入一个模拟的 IoT Hub 的消息：

```
{"deviceId": "TestDevice1","temperature": 30.337316254310924,"guid": "566
c8bad-325a-4cca-8a36-c204d322005f"}
```

点击“执行”按钮，可以看到日志窗口输出函数执行完毕。为了确认这个模拟的 IoT Hub 的消息是成功写入了之前创建的存储账号，需要安装一个应用程序 Azure Storage Explorer，下载地址为`http://storageexplorer.com/`。安装以后用存储账号名和主密码登录，会发现如图10.42所示有一条记录已经在 iotHub1storage 账号下面的 Tables/temperatureData 路径中。

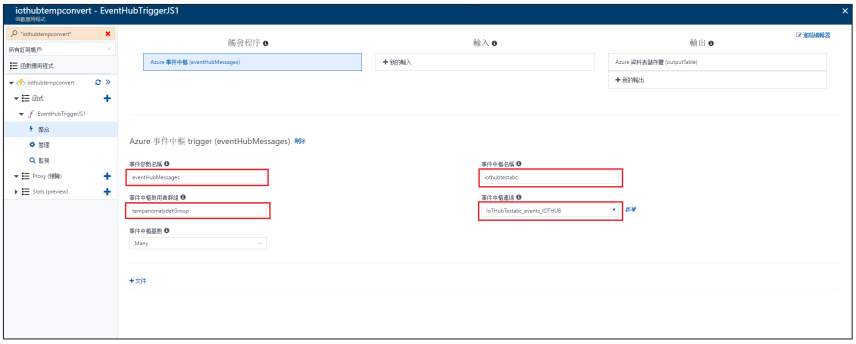


图 10.42　事件中枢

看到这条记录，说明 JavaScript 代码运行成功了。接下来可以用真实的 IoT Hub 消息来测试。

之前，我们写了一个 C# 程序 VirtualDeviceApp.cs 向 IoT Hub 发送消息，那个程序在温度高于 33 摄氏度的时候，发送的消息是“This is a high temperature”，而不是 Json 对象，所以需要做如下微小的修改：注销第 20 行 // messageString = "This is a high temperature"；另外，需要删除 IoT Hub 的路由，否则高于 33 摄氏度的消息会被送到 Service Bus 的 testQueue 队列。

重新编译运行。查看 Azure Storage Explorer 窗口，程序产生的消息显示了大量的新数据。图10.43展示了一个例子。

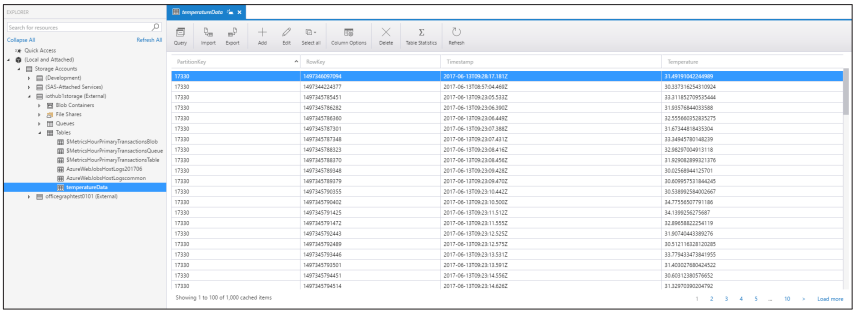


图 10.43　新数据示例

然后修改 Azure 机器学习工作室里的程序，把输入换成从存储账号读入，如图10.44所示。

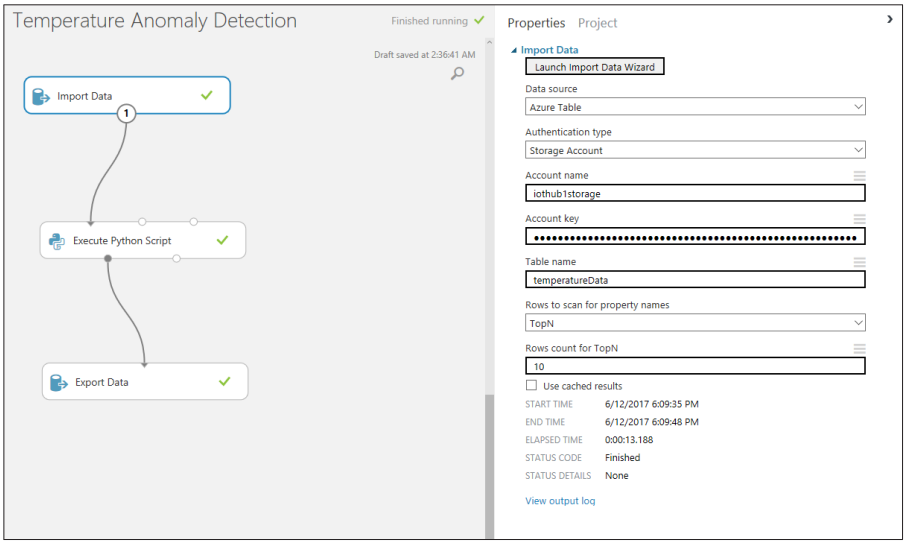


图 10.44　把输入换成从存储账号读入

点击“Export Data”选项，设置学习以后的输出路径。

这个设置把结果存在 iotHub1storage 的 BLOB 容器/iot/result.csv。运行程序完毕后，查看 Azure Storage Explorer 窗口，可以看到如图10.45所示的结果。

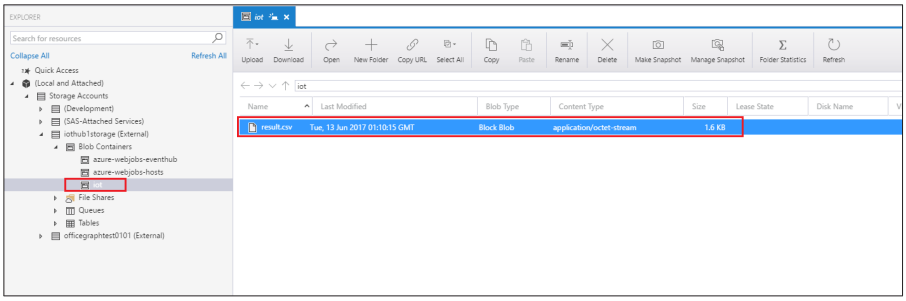


图 10.45　Azure Storage Explore 窗口

下载 result.csv 文件，用 Excel 内置画图工具可以很明显地看到被标记成警告的高温度。

最后，将这个机器学习的功能发布成网络服务。在上面这个例子里，输入和输出都是固定的。如果要把这个功能做成网络服务，就要把输入和输出做成通用的。所以就做出如下修改，并且像图10.46展示的一样作为一个网页发布。

出于验证的目的，可以在发布的网页上选择 csv 文件作为输入，这时会出现如图10.47所示的界面。

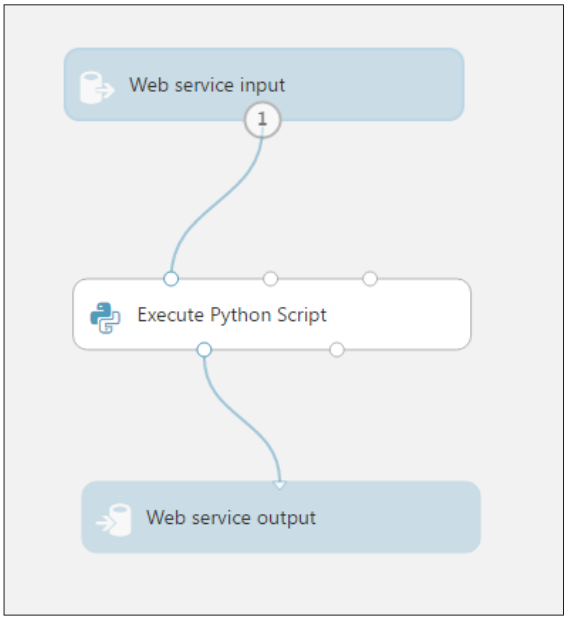


图 10.46　网页发布所需的服务结构

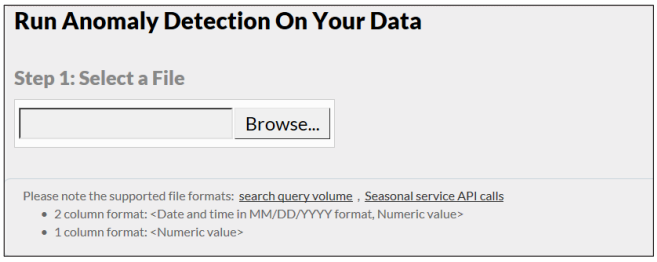


图 10.47　把输入输出作为网络服务发布

运行以后，会出现如图10.48所示的结果。可以很明显地看到，在 3 倍标准差以外的点都被标注成了异常（黑色圆点）。

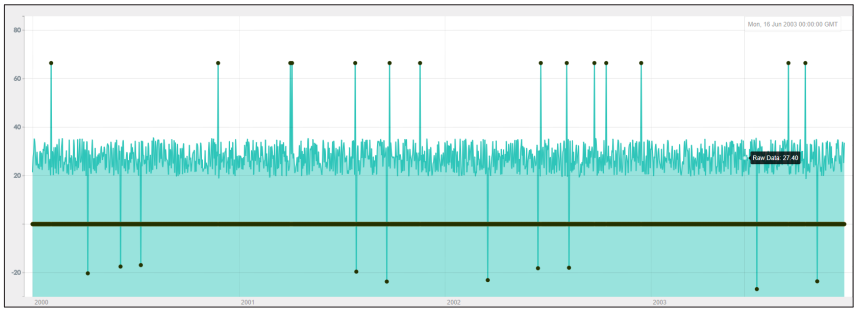


图 10.48　异常侦测结果